University of Pisa
Computer Science Department

Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns

by

Massimo Torquati

Dottorato di Ricerca in Informatica
Ph.D. Thesis

March 8th, 2019

Supervisor: Prof. Marco Danelutto

Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns

by

Massimo Torquati

Abstract

Multi-core computing systems are becoming increasingly parallel and heterogeneous. Parallelism exploitation is today the primary instrument for improving application performance. Despite the impressive evolution of parallel HW, parallel software development tools have not yet reached the same level of maturity.

Programmers need high-level parallel programming models capable from the one hand of reducing the burdens of the efficient parallelization of their applications, and from the other side of accommodating hardware heterogeneity. Abstracting parallel programming by employing parallel design patterns has received renovated attention over the last decade. However, the pattern-based approach to parallelism exploitation suffers from limited flexibility and extensibility.

In this thesis, we present the new version of the FastFlow parallel library that completes and strengthens the library that we started developing since 2010. Over the years the FastFlow pattern-based library was used in three EU-funded research projects. The lessons learned over this period of research and parallel software development convinced us to redesign, restructure and improve the lower level software layers of the library and also to introduce a new concurrency graph transformation component aiming at refactoring the parallel structure obtained through patterns compositions. The objectives of the new FastFlow design are twofold: *a)* to increase flexibility and composability of the approach while preserving its efficiency, and *b)* to introduce new features that open to the possibility of introducing static optimizations.

We propose a small set of highly efficient, customizable and composable *parallel building blocks* that can be connected and nested in many different ways and that provides the user with a reduced set of powerful parallel components. The base idea mimics the RISC approach of microprocessor architectures. The FastFlow library targets domain-expert programmers by offering some well-know high-level parallel patterns as well as run-time system programmers by providing a set of parallel building blocks along with clean and effective data-flow composition rules.

The new FastFlow software layer provides the essential mechanisms to restructure the data-flow concurrency graph produced by patterns and building blocks compositions. Straightforward yet effective graph transformations are transparently and automatically provided to the user through optimization flags. Its clean API enables the implementation of new and more powerful static optimization policies.

Finally, a full set of experiments is discussed assessing both functional and non-functional properties of both the building block set and the transformation rules.

To Marcella, Riccardo, Marta e Sara.

“Costanzia: non chi comincia, ma quel che persevera”

Leonardo da Vinci

Acknowledgments

First and foremost I thank Marco Danelutto for his guidance, support, and friendship.

I am especially thankful to Marco Aldinucci and Peter Kilpatrick not only for their friendship but also for the numerous and fruitful discussions we had together over the years and for the continued enthusiasm for our collaborative work.

My gratitude also goes to the people of the Parallel Computing Group with which I had the privilege of working. Among them, Gabriele Mencagli, Daniele De Sensi and Tiziano De Matteis deserve a special thank you.

Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

The slowdown of Moore's law and the end of Dennard scaling has produced a radical change in the computing landscape. We are witnessing a continuous increase in the number of cores packed in a single chip and to the on-chip integration of standard CPUs with specialized HW accelerators (e.g., FPGAs). Recently a new term "Megacore era" has been coined indicating that we are entering a new era of computing where "hundreds of thousands, or millions, of (small) cores" will be available to the programmer on a single or few highly heterogeneous devices [186]. Examples of current highly parallel general-purpose multi-core platforms are the Intel's Xeon Phi KNL featuring 72 hardware cores (4-way SMT) for a total of 288 logical cores (at 1.5GHz), and the IBM Power9 E980 with 24 hardware cores (8-way SMT) for a total of 192 logical cores (up to 4.0GHz). The Intel Xeon Gold 6138P Processor is an example of integration in a single chip of a high-end multi-core with a high-performance and power-efficient mid-range FPGA [98].

Today, the primary means for improving application performance is to employ parallel processing at the application level. The "free lunch" era of continuously increasing processor speeds has ended [307]. Despite the impressive evolution of the HW platforms, a wide gap still exists between parallel architectures and parallel programming maturity, and the software community is slowly moving to fill this gap. It is crucial, then, that the research community makes a significant advance toward making the development of parallel code accessible to all programmers achieving a proper trade-off among *Performance*, *Programmability* and *Portability.* The new "free lunch" requires scalable parallel programming [242].

In the parallel scientific community, it is widely recognized that *high-level parallel abstractions* are the solution to the "programmability wall" [85, 34], and they represent the cornerstone for achieving *performance portability* [241, 242, 94, 268, 66, 129]. A quantitative definition of *performance portability* has been recently proposed by Pennycook, et al. [269]. Informally, it can be defined as *the ability to compile the same code on different platforms maintaining the desired level of performance considering the features and the peak-performance of the target platform.* The challenge is to be able to guarantee adequate performance, possibly close to hand-tuned parallel implementation made by experts, without sacrificing *code portability* and *programmability.* This last aspect is particularly critical to enlarge the parallel software developer community to non-expert parallel programmers.

Abstracting parallel programming by employing parallel design patterns has received renovated attention over the last decade. The programming approach based on parallel patterns is denominated *structured parallel programming* [122, 123, 94, 242, 318]. This term has been borrowed from sequential programming where in the 60s and 70s programs were often poorly designed [139]. The foundational ideas were initially proposed in the late 80s when the *algorithmic skeleton* concept was introduced to simplify parallel programming [93, 267]. Briefly, algorithmic skeletons may be considered as a best practice of implementation of parallel patterns [88]. The key benefits of parallel patterns are that they abstract over many low-level implementation details merging parallelism exploitation and synchronization requirements at a sufficiently high level of abstraction. The definition of suitable parallel abstractions with their associated functional and parallel semantics requires efficient parallel Run-Time Systems (RTS, from now on), which in turn requires a deep understanding of all aspects of modern parallel architectures. Parallel patterns have been used in multiple recent research proposals [64, 134, 282, 43, 233].

However, one of the main limitations of concrete implementations of the parallel patterns approach is that it restricts the possibilities to express parallelism only to those patterns (and their compositions) offered by the framework/library considered. For this reason, some parallel software developers consider the parallel pattern-based approach as not enough flexible and extensible to be used in practice. Therefore, it is of paramount importance, when designing a new software stack inspired by this programming methodology, to provide the RTS programmer with suitable tools and mechanisms that allow him/her to build new parallel patterns and to integrate existing ones with *ad hoc* parallel solutions in a well-defined way. In this case, "well-defined" means that the extensions and customizations should not impair the overall efficiency and portability of the structured approach and that they can be integrated into the considered system as smoothly as possible. In that regard, some recent research works advocated that a relatively small number of parallel components can be used to model a broad range of operations and high-level parallel components covering multiple application domains efficiently [306, 7]. These results have opened a new perspective to overcome the flexibility limitation of the pattern-based approach.

The *structured parallel programming methodology* was the fertile ground in which the initial ideas of the FastFlow project have grown[1], then such programming methodology guided us during its implementation. The "FastFlow way" to harnessing parallel programming on multi/many-core platforms is the main subject of this dissertation. The lessons learned over about eight years of research and parallel software development using the library led us to recently *redesign and improve the lower level software layers of the* FastFlow *library to make it more powerful and flexible* and *to enrich the library with a new software component for managing concurrency graph transformations/optimizations.*

Thesis aims and context.

The main aim of the research effort described in this thesis is to provide a redesign of the FastFlow *parallel library capable of offering a reduced set of powerful parallel components mimicking the idea of the RISC-like approach in microprocessor architectures. We want to demonstrate that, by using such building blocks, it is easy to*

[1] The FastFlow project home is `http://calvados.di.unipi.it/fastflow`

build complex parallel patterns and to introduce concurrency graph transformations and optimizations capable of enhancing performance portability. The final goal is to substantiate that the performance obtained by the resulting parallel solutions targeting multi/many-core platforms are as good as or even better than state-of-the-art parallel frameworks, but with more programmer flexibility, enhanced performance portability and software engineering advantages.

We do believe that the FastFlow programming model based on the proper composition of parallel building blocks presents a clear methodology for building parallel components providing the RTS programmer with the right level of abstraction to develop efficient and flexible solutions.

The FastFlow project started around 2010 with the primary aim at targeting efficient parallel streaming computations on emerging multi-core platforms [22]. Since then, it has been extended and improved in different directions, from the targeting of distributed memory systems [11, 73], to the definition of OpenCL-based and CUDA-based heterogeneous patterns targeting CPU+GPUs [23, 8]. Since the beginning of the project, the author was (and still he is at the time of writing) the leading developer and the maintainer of the FastFlow library.

A significant boost to the development of the FastFlow library came from its utilization as RTS in three successful European research projects (EU-FP7 ParaPhrase, EU-FP7 REPARA, and EU-H2020 RePhrase). The usage of the FastFlow library within these EU-funded research projects, allowed us to enhance both its RTS and its usability [119, 108], and to actively contribute to the success of the projects. Besides, over the years, the FastFlow library has sustained several research works including some Ph.D. theses covering different aspects of multi-cores and many-cores programming [168, 240, 104, 250, 143, 128], and it has also been used for years as a tool to teach structured parallel programming at the *Master Degree in Computer Science and Networking* jointly run by the University of Pisa and Scuola Superiore Sant'Anna [120].

Since the initial phases of the FastFlow project, we firmly believed that a relatively small set of primitive parallel components (in this thesis called *building blocks*) could be used to model a wide range of parallel computations across multiple application domains [118, 7]. Also, the *stream* concept revealed itself as a powerful abstraction particularly suitable for modern multi-core platforms [18]. Therefore, *parallel building blocks* and *streaming* are the two fundamental ingredients of the FastFlow library.

Following the same principles of the structured parallel programming methodology, a parallel application (or a parallel component) is conceived by selecting and adequately assembling a small set of well-defined building blocks modeling both data and control flows. Parallel building blocks are key elements that can be used by the RTS developer to construct high-level parallel patterns as well as by the application programmer to design his/her parallel components for implementing specific parts of the application at hand.

The idea is to promote the so-called *LEGO-style* approach to parallel programming where the bricks can be either complex pre-assembled and tested structures (i.e. parallel patterns) or elementary bricks (i.e. *parallel building blocks*). As we shall discuss in Chapter 5, the FastFlow parallel building blocks are: the *pipeline* modeling data-flow pipeline parallelism, the *farm* modeling functional replication coordinated by a centralized entity, and the *all-to-all* building block modeling both functional replication without a centralized coordination and the shuffle communication pattern between function replicas. These parallel building blocks can be combined and nested in many ways forming either acyclic or cyclic *concurrency graphs*, where nodes are FastFlow concurrent entities and edges are communication channels.

The idea of modeling parallel applications as compositions of basic building blocks can be considered at the core of the structured parallel programming methodology. It promotes the pattern-based approach not only at the application level to ease the application programmer's job but also at a lower level of abstraction, that is, in the design and implementation of the underlying RTS by providing suitable parallel abstractions capable of simplifying the job of the RTS developer [7].

In addition to the *composability* property of the LEGO-style approach, the FastFlow building blocks promote *specialization* as a way to create custom and well-defined new building blocks, which seamlessly integrate into the FastFlow application topology. For example, the FastFlow *farm* building block may be customized with *ad hoc* input data scheduling and output results gathering policies to fulfill specific application needs. Such operation is straightforward to implement, and it is confined to the single building block considered with no intrusion within the FastFlow RTS.

Given a FastFlow concurrency graph describing a given parallel application, the LEGO-style approach to parallel programming allows to deconstruct the graph easily and to reassemble the building blocks taken apart in a different and possibly better way according to some optimization objectives. Examples of possible optimizations are: to reduce the number of resources used by the application (i.e. to increase its efficiency); to remove potential bottlenecks introduced by the composition and nesting of high-level parallel patterns (i.e. to improve the application performance). This approach opens many opportunities to devise and introduce smart optimization strategies on top of the FastFlow building block layer, for example, through a Domain Specific Language (DSL) interpreter [121, 163], or by implementing higher-level frameworks such as PiCo [251] and Nornir [131, 131], or DSL compilers such as SPar [177]. The primary objectives of the optimizations are to improve both the *performance* of applications and also to enhance their *performance portability* on different target platforms.

To this end, we introduced a new software layer in the FastFlow software stack called *graph transformation component*, which provides the essential mechanisms to operate directly at the building block level to restructure the application concurrency graph to accommodate the efficient execution of FastFlow applications on different target platforms. A small set of already defined transformations (e.g., to remove adjacent service nodes in the network topology) and a set of predefined combining functions working with parallel building blocks (e.g., merging two or more *farm* components connected in *pipeline*) have been designed and provided as ready-to-use transformations to the FastFlow programmer. Some of these graph transformations can be automatically enabled by setting proper optimization flags before starting the execution of the FastFlow parallel program.

Thesis Contributions

The main contributions of this dissertation, that complete and improve the FastFlow library as previously developed by the author, are as follows:

C1 *A comprehensive and thorough description of the* FastFlow *library design, its parallel programming model and its implementation.* Although several research papers describe and discuss features of the FastFlow library, no organic description of the FastFlow programming model and its implementation was previously stated.

C2 *The definition and implementation of the new building blocks layer in the* FastFlow *software stack.* In this layer, a new parallel component called *all-to-all* modeling the *shuffle* communication pattern, and a set of new composition and combining rules have been defined, which enable the construction of complex non-linear streaming networks made of both *sequential nodes*, i.e. *standard*, *multi-input*, and *multi-output nodes*, as well as *parallel building blocks*, i.e. *farm*, *all-to-all* and *pipeline*. The latter, coordinates the parallel execution of a proper assembly of a well-defined set of sequential nodes. The new *all-to-all* parallel building block can also be profitably used to optimize typical compositions of patterns such as the widely used *Map+Reduce* compositions or *farm+farm* compositions where the two farms have a different number of Workers. The network topology describing the parallel application is obtained by a data-flow composition of building blocks connected by bounded or unbounded FIFO channels implemented by using Single-Produce Single-Consumer (SPSC) FIFO queues. The unbounded FastFlow SPSC queue is considered state-of-the-art and is one of the distinguishing features of the library [19]. We designed the FastFlow channels at the beginning of the project [312]. **Related publications:** [7, 118].

C3 *A new* FastFlow *concurrency graph transformation software component.* This component allows to statically (and automatically) introduce graph transformations capable of optimizing the FastFlow concurrency graph describing the application. The transformations proposed mainly aim to reduce both the number of nodes implementing the data-flow network by removing “unnecessary” fan-in/fan-out service nodes and to optimize particular pattern compositions (e.g., compositions of *Map* and *Reduce*). The objective is on the one hand to increase resource efficiency and on the other hand to promote performance portability on machines with a different number of computing resources. **Related publication:** [129].

C4 *The implementation of both blocking and non-blocking concurrency control mechanisms for accessing the communication channels* connecting two concurrent entities in the run-time. The proposed mechanisms allow us to improve power saving and/or throughput by statically and dynamically switching the concurrency mode between passive waiting (blocking) and active waiting (non-blocking) of run-time threads. This is particularly relevant for long-running data streaming computations where variable arrival rates and sudden workload changes require different levels of reactiveness to respect a given QoS level. **Related publication:** [314].

C5 *The implementation of concurrency throttling mechanisms in the farm building block*. This enables the development of sophisticated policies that dynamically change the concurrency level of parts of the parallel application (e.g., in a *farm* building block) to increase either the sustained input rate in a pipeline computation or to reduce the power consumption by reducing the number of Worker threads. **Related publications:** [114, 130].

C6 *Experimental validation of the proposed building blocks to support the efficient implementation of well-known parallel patterns.* We aimed at demonstrating that FastFlow building blocks can also be profitably used for implementing usable and efficient task-based patterns (such as *Divide & Conquer*, *Macro Data-Flow* and *ParallelFor*) with the same level of performance and usability of specialized library using a task-based programming model such as Intel TBB and the OpenMP standard. **Related publications:** [109, 119, 69].

Thesis outline

This thesis is organized in nine chapters. Apart from the first and last chapters presenting the introduction of this work and outlining the conclusions and possible future research directions, the other chapters are organized as follows:

Chapter 2: It provides the essential ingredients to help the reader going through the contributions of the thesis and provide the relevant background to comprehend related research works. We give an overview of the complex and variegated world of parallel computing focusing on the consequences of the transition from multiprocessors to multi/many-core systems. Finally, we describe the most relevant programming models focusing on the *structured parallel programming methodology* which is at the basis of our work.

Chapter 3: In this chapter, we present the frameworks and libraries targeting parallel programming on multi-cores with particular focus on frameworks promoting high-level parallel programming that are close to the FastFlow approach. The main aim of this chapter is to provide the reader with a broad spectrum of past and current research efforts in the context of structured parallel programming and parallel programming in general.

Chapter 4: This is the central chapter of the thesis. It presents an overview of the FastFlow library, its programming model, the distinguishing features of the new version compared to previous versions of the library and also the new layered software design. We briefly go through the FastFlow project history starting from its first version released in 2010 and then we highlight the most significant enhancements it had over about eight years of development. This chapter, provides the reader with the links to other chapters where a broader discussion of the most important aspects can be found. **Contribution C1**.

Chapter 5: This chapter presents the FastFlow building blocks, their composition rules, and their parallel semantics. FastFlow building blocks promote a LEGO-style approach to parallel programming targeting mainly RTS programmers rather than application programmers. The idea underpinning *building block* is that of promoting the structured parallel programming methodology at a lower level of abstraction compared to the one promoted by parallel patterns. The aim is to make available suitable parallel abstractions to RTS developers so that they can develop efficient and portable high-level parallel components. **Contribution C2**.

Chapter 6: In this chapter we describe the FastFlow communication channels which play a crucial role both for performance and parallel execution correctness reasons. A communication channel connecting two FastFlow concurrent entities is implemented by using a First-In First-Out (FIFO) Single-Producer Single-Consumer (SPSC) queue. This kind of queues are particularly interesting because they can be implemented in a very efficient way on shared-cache multi-cores. The FastFlow channel has been implemented to support both *blocking* and *non-blocking* concurrency control policies and provides the necessary hooks allowing the automatic switching between the two concurrency modes dynamically. This chapter describes the two concurrency control policies used to regulate concurrent accesses to the channel and used for improving performance and power efficiency. **Contributions C4**.

Chapter 7: This chapter elaborates on the implementation of the *building blocks* (both sequential and parallel ones). The FastFlow *node* concept and its implementation is presented. It represents the atomic concurrent entity provided by the FastFlow library. The *sequential node combiner* as well as all different kind of sequential nodes are presented. The *pipeline*, *farm* and *all-to-all* building blocks are discussed presenting their features and capabilities and assessing the parallel overhead they introduce in FastFlow applications. This chapter also discuss the implementation of the dynamic *concurrency throttling* of *farm*'s Workers. **Contribution C2, C5**.

Chapter 8: This chapter presents the new concurrency graph transformation software component added to the FastFlow software stack. This layer contains a set of mechanisms and functions that allow us to statically restructure the application concurrency graph to reduce the number of FastFlow nodes and to optimize particular combinations of building blocks. A simple interface function is provided to the user to automatically apply simple yet powerful transformations. More complex graph transformations based on the provided mechanisms are currently user's responsibility. **Contribution C3**.

Chapter 9: In this chapter, we discuss some parallel patterns implemented on top of FastFlow building blocks. Notably we considered: the *ParallelFor* pattern providing a versatile implementation of the *Map* and *Map+Reduce* abstractions; the *Macro Data-Flow* pattern modeling general, non-recursive, parallel computations by automatically managing data-flow dependencies among tasks; and finally, the *Divide & Conquer* parallel pattern modeling classical recursive computations. We assess both usability and performance of these three high-level patterns by using simple benchmarks as well as real applications/kernels and comparing performances with state-of-the-art framework offering either native construct (i.e. parallel-for) or general implementation mechanisms such as the ones provided by task-based programming model. **Contribution C6**.

Possible ways to read the thesis

There is more than one possibility for reading this thesis apart from barely following the numerical order of chapters straight from the Introduction to the Conclusion.

Chapter 4 has been conceived as the central chapter of the thesis, summarizing all works done over the years related to the FastFlow parallel programming library we developed, and what we have added to the library during the Ph.D. period. It also discusses the *programming model* offered by the library, a fundamental concept that is of primary importance to understand the contributions presented in technical chapters, and introduces the primary aspects discussed in more detail in other chapters. The reader who is familiar with parallel programming approaches and particular with structured parallel programming, might consider to skip the background and related work chapters (i.e. Chapter 2 and Chapter 3, respectively) and read Chapter 4 after the Introduction chapter. In any case, by considering the connections between chapters reported in Figure 1-1 (thin solid gray lines), the reader may decide to follow

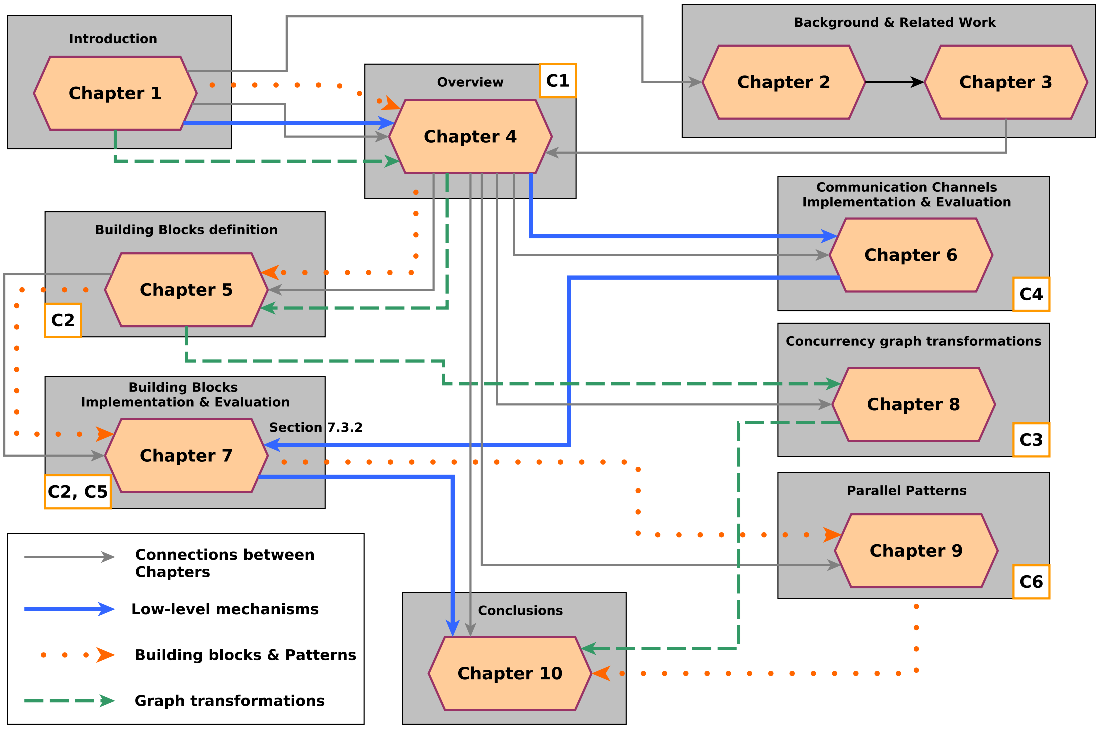


Figure 1-1: Possible ways to read this dissertation.

his/her preferred path. Among the possible paths, and apart from Chapter 1 and 10, we would like to highlight three paths which have a particular focus, notably: Low-level mechanisms, Building blocks and Patterns and Graph's transformations.

- **Low-level mechanisms.** It includes: Chapters 4, 6, and Section 7.3.2 of Chapter 7 (thick solid blue lines). This path allows consideration of contributions **C1**, **C4** and **C5** focusing on low-level mechanisms: communication channels implementation, *concurrency control* policies and the *concurrency throttling* mechanisms.

- **Building blocks and Patterns.** It includes: Chapters 4, 5, 7, 9 (dotted red lines). This path allows consideration of contributions **C1**, **C2**, and **C6** skipping all details about the implementation of channels and concurrency graph transformations.

- **Graph's transformations.** It includes: Chapters 4, 5, 8 (dashed green lines). This path allows consideration of contributions **C1**, **C2**, **C3** skipping the details related to the channels, building blocks and pattern implementations.

Publications pertaining to the topics of the thesis

In this section, we summarize the list of the author's publications pertaining to the topics covered in the thesis divided by journal and conference publications. Although some of the listed publications do not directly refer to specific contributions included in the present thesis, all of them acted as either inspiration, preliminary study, or interesting use cases used also to test the FastFlow library features. For this reason, in the following, we report all of them in ascending order on the basis of the year of publication or on the year of when they appeared on-line if "In Press".

Journal publications

1. M. Torquati, D. De Sensi, G. Mencagli, M. Aldinucci, and M. Danelutto. "Power-aware pipelining with automatic concurrency control." *Concurrency and Computation: Practice and Experience*, 2018. (In Press). DOI: 10.1002/cpe.4652
2. G. Mencagli, M. Torquati, and M. Danelutto. "Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams." *Future Generation Computer Systems*, 79:862 – 877, 2018. DOI: 10.1016/j.future.2017.09.004
3. G. Mencagli, M. Torquati, F. Lucattini, S. Cuomo, and M. Aldinucci. "Harnessing sliding-window execution semantics for parallel stream processing." *Journal of Parallel and Distributed Computing*, 116:74 – 88, 2018. DOI: 10.1016/j.jpdc.2017.10.021
4. M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, M. Torquati, M. Aldinucci, and P. Kilpatrick. "The RePhrase Extended Pattern Set for Data Intensive Parallel Computing." *International Journal of Parallel Programming*, 2017. (In Press). DOI: 10.1007/s10766-017-0540-z
5. M. Torquati, G. Mencagli, M. Drocco, M. Aldinucci, T. De Matteis, and M. Danelutto. "On Dynamic Memory Allocation in Sliding-Window Parallel Patterns for Streaming Analytics." *The Journal of Supercomputing*, 2017. (In press). DOI: 10.1007/s11227-017-2152-1
6. M. Danelutto, P. Kilpatrick, G. Mencagli, and M. Torquati. "State access patterns in stream parallel computations." *The International Journal of High Performance Computing Applications*, 2017. (In press). DOI: 10.1177/1094342017694134
7. M. Danelutto, D. De Sensi, and M. Torquati. "A power-aware, self-adaptive macro data flow framework." *Parallel Processing Letters*, 27(1):1–20, 2017. DOI: 10.1142/S0129626417400047
8. D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto. "Bringing parallel patterns out of the corner: The P^3ARSEC benchmark suite. *ACM Trans. Archit. Code Optim.*, 14(4):33:1–33:26, 2017. DOI: 10.1145/3132710
9. D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, J. D. Garcia, M. Danelutto, and M. Torquati. "Finding parallel patterns through static analysis in C++ applications." *The International Journal of High Performance Computing Applications*, 2017. (In Press). DOI: 10.1177/1094342017695639
10. M. F. Dolz, D. Del Rio Astorga, J. Fernandez, M. Torquati, J. D. Garca, F. Garca-Carballeira, and M. Danelutto. "Enabling semantics to improve detection of data races and misuses of lock-free data structures." *Concurrency and Computation: Practice and Experience*, 29(15), 2017. (In press). DOI: 10.1002/cpe.4114

11. D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. “SPar: A DSL for high-level and productive stream parallelism.” *Parallel Processing Letters*, 27(01):1740005, 2017. DOI: 10.1142/S0129626417400059
12. G. Mencagli, M. Torquati, M. Danelutto, and T. De Matteis. “Parallel continuous preference queries over out-of-order and bursty data streams.” *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2608–2624, 2017. DOI: 10.1109/TPDS.2017.2679197
13. M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. “A parallel pattern for iterative Stencil + Reduce.” *The Journal of Supercomputing*, 2016. (In Press) DOI: 10.1007/s11227-016-1871-z
14. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. “Pool Evolution: A Parallel Pattern for Evolutionary and Symbolic Computing.” *International Journal of Parallel Programming*, 44(3):531–551, 2016. DOI: 10.1007/s10766-015-0358-5
15. M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. “Data stream processing via code annotations.” *The Journal of Supercomputing*, Jun 2016. (In Press). DOI: 10.1007/s11227-016-1793-9
16. M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati. “Parallel Visual Data Restoration on Multi-GPGPUs using Stencil-Reduce Pattern.” *The International Journal of High Performance Computing Applications*, 29(4):461–472, 2015. DOI: 10.1177/1094342014567907
17. S. Campa, M. Danelutto, M. Goli, H. Gonzlez-Vlez, A. M. Popescu, and M. Torquati. “Parallel patterns for heterogeneous cpu/gpu architectures: Structured parallelism from cluster to cloud.” *Future Generation Computer Systems*, 37:354 – 366, 2014. DOI: 10.1016/j.future.2013.12.038
18. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. “Design patterns percolating to parallel programming framework implementation.” *Int. J. Parallel Program.*, 42(6):1012–1031, 2014. DOI: 10.1007/s10766-013-0273-6
19. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. “Targeting heterogeneous architectures via macro data flow.” *Parallel Processing Letters*, 22(2), 2012. DOI: 10.1142/S0129626412400063

Conference/Workshop publications

1. D. De Sensi, P. Kilpatrick, and M. Torquati. “State-Aware Concurrency Throttling.” In *Proceedings of International Parallel Computing Conference (ParCo 2017)*, Advances in Parallel Computing, pg. 201–210, 2018. DOI: 10.3233/978-1-61499-843-3-201
2. M. Danelutto and M. Torquati. “Increasing efficiency in parallel programming teaching.” In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pg. 306–310, 2018. DOI: 10.1109/PDP2018.2018.00053
3. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. “FastFlow: High-Level and Efficient Streaming on multi-core.” In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, Chapter 13. John Wiley & Sons, Inc, 2017. (accepted in 2012). DOI: 10.1002/9781119332015.ch13
4. M. Danelutto, T. De Matteis, D. De Sensi, and M. Torquati. “Evaluating Concurrency Throttling and Thread Packing on SMT multicores.” In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pg. 219–223, 2017. DOI: 10.1109/PDP.2017.39
5. M. Danelutto, M. Torquati, and P. Kilpatrick. “A DSL based toolchain for design space exploration in structured parallel programming.” *Procedia Computer Science*, 80:1519 – 1530, International Conference on Computational Science 2016, ICCS 2016. DOI: 10.1016/j.procs.2016.05.477
6. M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. “A Divide-and-Conquer parallel pattern implementation for multicores.” In *Proceedings of the 3rd International Workshop on Software Engineering for*

Parallel Systems, SEPS 2016, pg. 10–19, 2016. DOI: 10.1145/3002125.3002128

7. M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. “Introducing Parallelism by Using REPARA C++11 Attributes.” In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pg. 354–358, 2016. DOI: 10.1109/PDP.2016.115
8. M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. “Parallelizing High-Frequency Trading Applications by using C++11 Attributes.” In *Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara)*, pg. 140–147, 2015. DOI: 10.1109/Trustcom.2015.623
9. M. Danelutto, D. De Sensi, and M. Torquati. “Energy driven adaptivity in stream parallel computations.” In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pg. 103–110, 2015. DOI: 10.1109/PDP.2015.92
10. M. Danelutto and M. Torquati. “Structured parallel programming with “core” FastFlow.” In V. Zsók, Z. Horváth, and L. Csató, editors, *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, 2013, Revised Selected Papers*, pg. 29–75. Springer International Publishing, Cham, 2015. DOI: 10.1007/978-3-319-15940-9_2
11. M. Danelutto and M. Torquati. “Loop parallelism: A new skeleton perspective on data parallel patterns.” In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pg. 52–59, 2014. DOI: 10.1109/PDP.2014.13
12. M. Danelutto and M. Torquati. “A RISC building block set for structured parallel programming.” In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pg. 46–50, 2013. DOI: 10.1109/PDP.2013.17
13. D. Buono, M. Danelutto, S. Lametti, and M. Torquati. “Parallel patterns for general purpose many-core.” In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pg. 131–139, 2013. DOI: 10.1109/PDP.2013.27

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background

2.1 Parallel Computing

The unquenchable need of scientists to perform more and more extensive computations has nurtured the evolution of computers and applications in the past fifty years. The only approach that was able to satisfy such scientific needs is *parallelism*, i.e. the capability of computing more than one task simultaneously.

A parallel computer is a "collection of processing elements that communicate and cooperate to solve a large problem fast" [28]. This definition at first glance appears straightforward, but it hides many complex problems that fed extensive scientific research. To mention some of them: How large does the "collection of processing elements" need to be to solve a problem? How processing elements "communicate" one each other and how frequently? How "cooperation" has to be handled? Answering these questions even for a single well-defined problem is not easy.

In principle, building a parallel computer to solve a given problem is easy to put into practice. It is sufficient to get a set of independent computers, each one possibly equipped with multiple execution units, connected by a communication network and using them simultaneously to solve a computation problem. The problem to be solved is broken down into independent discrete parts in a manner that the splitting does not impair the overall results; if possible, each part is further broken down into a series of instructions. Instructions coming from the different parts are then

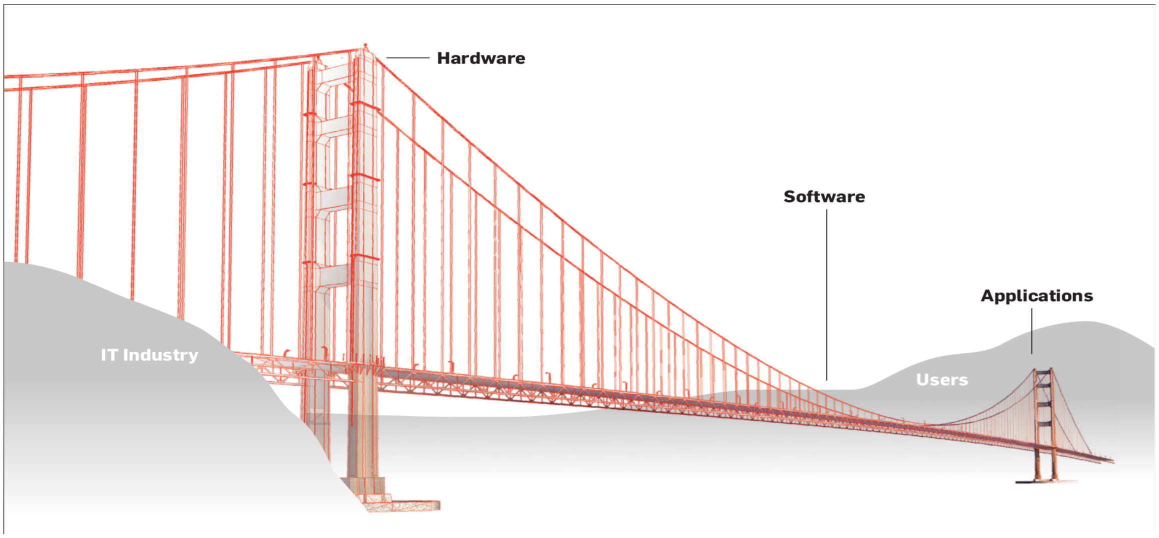


Figure 2-1: Bridge analogy connecting users to a parallel IT industry. From “A View of the Parallel Computing Landscape” [35].

executed simultaneously on different computational units of each computer. A global coordinator is used to control and regulate different phases of the computation and to assemble the partial results into the final one.

Doing this is not difficult per se, what instead is far more challenging is to build a well-balanced parallel system so that its exploitation is not hindered by any internal bottlenecks. In the previous naïve model of a parallel system, the coordinator and/or the network system might result to be the bottleneck making the overall performance of the system even worse than the one of the single computer unit composing the system. However, even if there are no serious bottlenecks in the system, an important limiting factor might be the way the initial problem is broken down into parts and how these parts are mapped onto the available computing resources. In the end, the huge research efforts of the last fifty years in academic and industrial settings demonstrated that the primary problem in parallel computing is *software*, not hardware.

This important issue was clearly identified as “the problem” in the famous Berkeley Report “The Landscape of Parallel Computing Research: A View from Berkeley” [34] in 2006, at the beginning of the “multi-cores era”. A few years later, the same authors stated: “*Software is the main problem in bridging the gap between users and the parallel IT industry. Hence, the long distance of the span in Figure 2-1 reflects* *the daunting magnitude of the software challenge*" [35].

Parallel programs are far more challenging to design, write, debug, and tune than sequential ones. Splitting the program into multiple parts making them run concurrently, introduces several new problems that programmers must tackle. The allocation of different parts of the program together with the fine-tuning of communications and synchronizations among concurrent parts are considered one of the greatest difficulties in parallel programming [244]. These aspects, if not properly handled, might undermine the overall parallelization thus lowering or nullifying the parallelization outcomes. This is due to the fact that, generally, more work needs to be done by the parallel application with respect to its sequential counterpart. This extra work, called "overhead", is introduced by parallel programming and it is associated with the execution of any non-functional code of parallel applications.

To quantify the impact of the overhead and to give an idea of the difficulties in the parallelization process, let us consider one of the examples reported in the reference book "Parallel Computer Architecture: A Hardware/Software Approach" by D. Culler et al. [100]. Figure 2-2 shows the speedup of the parallelization of the AMBER code on a 128-processor Intel Paragon machine for three distinct versions developed over a period of about six months. The initial parallelization of the AMBER code (version 8/94) showed good speedup only for a small number of processors. The second version (version 9/94) improved the scaling by optimizing the workload balancing among computing resources. The final effort to optimize communication (version 12/94), improved the speedup of the application significantly.

The overhead introduced by the parallelization is strictly related to the tools and programming models used. The scarcity of good high-level parallel programming libraries and environments often forces the application developers to rewrite the sequential programs into a parallel one trying to exploit all the (low-level) features and peculiarities of the target system. This makes difficult and time-consuming the development of efficient parallel applications, making also the software not easily portable with the same or better performance on newer architectures. This last aspect, often referred to as *performance portability* (see also Section 2.4.1) is particularly critical

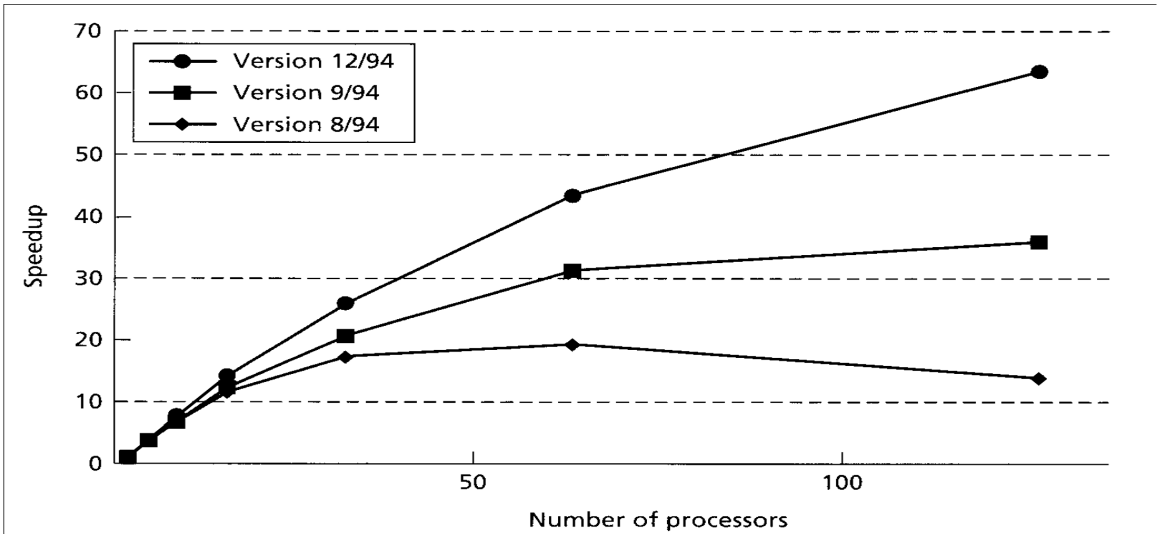


Figure 2-2: Speedup of three parallel versions of the AMBER code running on a 128-processor Intel Paragon machine. From “Parallel Computer Architecture: A Hardware/Software Approach” [100].

because of the rapid evolution of modern platforms. In the past few years, parallel computers have entered mainstream with the advent of multi-core computers. This makes urgent to define useful abstractions that allow easy and efficient development of parallel applications on the large set of systems equipped with different number of cores and different HW architecture. The applications running on multi-cores systems are not necessarily (parts of) large HPC applications but they are also standard software needing to fully use the available resources.

The advent of multi-core processors has alleviated several problems that are related to single-core processors (as for example the so-called *memory wall* [325]) but it raised the issue of the so called *programmability wall* [85]. Current approaches to exploit multi-core parallelism mainly rely on sequential flows of control (threads), and the sharing of data is coordinated by using locks (in shared memory systems) or explicit messages (in distributed memory systems).

It is widely acknowledged that *high-level parallel abstractions* are the solution to the programmability wall providing the ground for achieving *performance portability* [241, 242, 94, 268, 318]. Raising the level of abstraction in developing parallel applications can greatly simplify parallel programming by hiding all low level aspects related to task scheduling, resource mapping, synchronization and so on. The challenge is, at the same time, to be able to guarantee adequate performance, possibly close to hand-tuned parallel implementation made by experts, and most of all guarantee *performance portability* across different platforms.

Abstracting parallel programming by means of parallel design patterns have received lots of attention over the last decade. Parallel patterns are used in several recent systems [64, 134, 282, 43, 233]. The key benefit of parallel patterns is that they abstract over the low-level implementation differences merging parallelism exploitation and synchronization requirements at a sufficiently high level of abstraction. The definition of suitable parallel abstractions with their associated properties requires to be able to provide efficient parallel run-time systems, which in turn requires a deep understanding of all aspects of modern parallel architectures. Some recent research works have shown how a relatively small number of parallel components can be used to model a wide range of operations across multiple domains [306, 7].

In our opinion, further research is needed in this respect to fully recognize which are the most suitable abstractions for reasoning about parallelism that are able to play a key role in the scalable exploitation of ubiquitous parallelism.

2.2 The multi-core era of computing

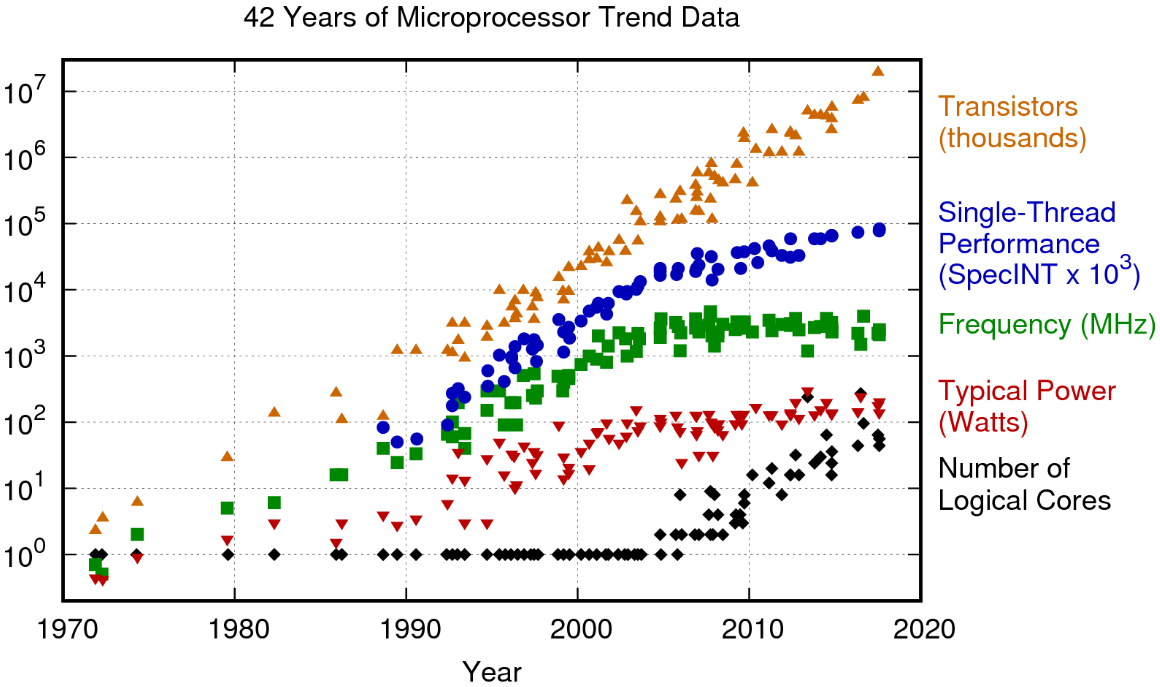
For almost forty years computing hardware has continuously evolved to sustain the high demand for increasing performance. From the early 70s up to 2004, we profited from the continuous growth of the performance capability of computer systems. This was mainly due to the success in the advance of the VLSI technology, which allows clock rates to increase, and also to pack more and more components into a single chip. More components in a chip translate in the execution of more instructions in parallel, and this, together with higher clock rates translates into increasing application performance without the need to touch even a single line of code. Such trend was predicted in 1965 by Gordon Moore, one of the founders of Intel, and it is known as *Moore's Law* [253], which roughly states that integrated circuit resources double every 18–24 months. Moore's law was the key indicator of successful leading-edge semiconductor products and companies at least until the mid-2000's.

The basic recipe for technology scaling was laid down by Robert N. Dennard at IBM in the early 1970s. This recipe is known as Dennard Scaling [137] and remained valid for about three decades. In a nutshell, the Dennard Scaling law states that as transistors get smaller their power density stays constant. Decreasing a transistor's linear size by a factor of two reduced power by a factor of four, or with both voltage and current halving [281].

Historically, the transistor power reduction afforded by the Dennard scaling law allowed manufacturers to drastically raise clock frequencies from one generation to the next one without significantly increasing overall circuit power consumption. If we consider the Intel 8086-8 processor at 8MHz and the Intel PIII at 1GHz, the first one had a power draw of about 1.8Watts whereas the PIII's power draw was about 33Watts, meaning that CPU power consumption increased by 18.3 times while CPU frequency improved by 125 times in about twenty years. Moreover, in the same time span, other fundamental technological advances occurred such as the adoption of L2 integrated caches, out-of-order instructions execution, superscalar and pipelining processors, all of them together boosted the computing power of the single computing system.

Although transistor size still continues to shrink, transistor power consumption no longer decreases accordingly. This has led to the end of the Dennard Scaling [89]. The primary reason for the breakdown of the Dennard scaling is that at small sizes, current leakage poses significant challenges, also causing the chip to heat-up with critical cooling problems. The breakdown of Dennard scaling and resulting inability to increase clock frequencies has caused most CPU manufacturers to stop trying to make the single processor run faster.

Around 2004, chip manufacturers started focusing on multi-core architectures as an alternative way to improve performance. Processor engineers, instead of boosting *Instruction Level Parallelism* and increasing clock frequency, started adding multiple processors (called *cores*) in a single chip making them communicate through shared



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Figure 2-3: Forty-two years of microprocessor evolution. Figure by K. Rupp [288].

hardware caches. The result was a Chip Multiprocessor, also referred to as CMP, in which the computing power can be fully exploited only through application parallelism, i.e. employing multiple cores to work on a single task. Since then, we entered the so-called "multi-core era" [195], where parallelism, at multiple levels of abstraction, is the driving force of computer design across all classes of computers, from small devices and desktop workstations to large-scale computer systems.

Figure 2-3 shows forty-two years (from about 1975 to 2017) of microprocessor evolution considering different metrics: 1) the number of integrated transistors; 2) single thread performance measured considering a well-known benchmark; 3) processor frequency; 4) power consumption; 5) and the number of logical cores (i.e. considering also hyperthreaded and SMT cores). It is clear from the plot that frequency and power do not experience any significant changes in the last ten years. Single-thread performance has kept increasing even if at a lower speed and this is mainly due to the introduction of smart power management and dynamic clock frequency scaling techniques. Moore's Law is still valid, in fact, for each year, the number of transistors

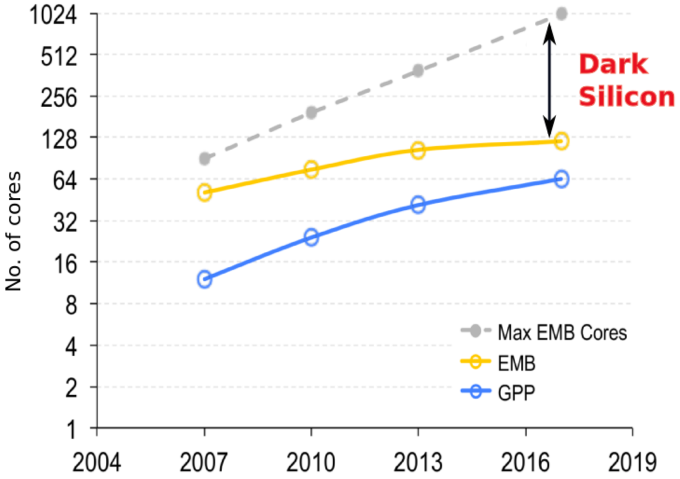


Figure 2-4: Number of general-purpose cores (GPP) and simpler embedded cores (EMB) that can be integrated on a die over different technology generations based on the available power budget. Figure from "Server Architecture for the Post-Moore Era" HotDC Keynote talk by Prof. B. Falsafi [152]. Full study presented in [188].

fitting into the same space, has continued to increase almost at the same rate. The number of logical cores per chip is increasing proportionally to the number of transistors. At the time of writing, a top-of-the-line Intel server processor has 56 logical cores (28 physical ones) while the Intel Knights Landing Xeon Phi CMP is equipped with up to 288 logical cores (72 physical ones).

2.2.1 Dark Silicon

The broad diffusion of CMP architectures has had and still is having, an important effect on how software is developed. The "free lunch era" of continuously increasing single processor speeds ended around 2005 [307]. Since then, the old software will run slower on new machines and an increase of performance can only be obtained if the application can run in parallel over the available cores. Advances in technology will mostly mean an increased number of cores. The full exploitation of such parallelism is one of the outstanding challenges of modern Computer Science. Packing multiple processors on a single chip does not automatically guarantee lower power consumption, instead, this technology enables the continuous improvement of the performance by replacing a high-clock-rate with efficient exploitation of the application parallelism.

However, if from one side increasing the core count allows increasing overall peak performance leveraging on parallelism, from the other side the increased number of active elements coming from multiple cores and large shared cache still results in increased overall power consumption. Power consumption together with limited cooling technology and device packaging constraints produced the so-called “dark silicon” problem [151, 150]. That is, a significant amount of transistor on-chip cannot simultaneously be powered-on at the peak performance level and thus stay “dark”, i.e. in low-power states.

Figure 2-4 shows the number of general-purpose high-performance high-power cores (GPP) and simple low-power embedded cores (EMB) that can be integrated on a die over multiple technology generations on the basis of the available power budget. In the figure, it is also reported the theoretical maximum number of embedded cores that could fit in the chip area. As can be seen, considering the available power budget and the peak performance attainable using GPP cores, a large fraction of chip area is dark due to power and thermal constraints. The main reason stands on the fact that up to half of the on-chip area is dedicated to large caches needed to sustain the memory bandwidth required to reach the desired performance [188]. However, such large caches are often ineffective at improving the performance of a wide range of real server workloads [154].

The switch from homogeneous architecture designs to heterogeneous ones provides more flexibility in utilizing the available power budget. On the other hand, complex heterogeneous multi-cores, presents even harder difficulties from the programming standpoint if compared to homogeneous multi-cores. At present, each computing element (CPU, GPU, DSP, FPGA) follows its own programming model and these divergent programming models are one of the most crucial obstacles to the broader acceptance of heterogeneous systems. The OpenCL programming standard [180], is a recent development effort promoted by several hardware vendors in the direction of enhancing program portability across CPU, GPU, and FPGAs. However, a unified software tool-chain for heterogeneous multi-cores remains one of the main challenges in Computer Science

2.3 Parallel Systems Architectures

One of the possible categorizations of parallel hardware is based on the number of instruction streams and the number of data streams [266]. This classification was proposed by Michael J. Flynn in 1966 and it is still used today [156]. A conventional uniprocessor system has a “Single Instruction stream and Single Data stream” (SISD). A conventional multiprocessor has “Multiple Instruction stream and Multiple Data stream” (MIMD). MISD (“Multiple Instruction stream and Single Data stream”) architectures are not popular today. In these systems, many functional units perform different instructions on the same data set. These types of architecture are mainly used for fault-tolerant purposes where the same instructions are executed redundantly to handle faults and detect errors.

Parallelism at instruction level is exploited by SISD architectures, whilst MIMD architectures leverage parallelism at the process level. In SIMD architectures the same instruction is executed by multiple processors on different data streams. They support data-level parallelism by applying the same operations to multiple items of data in parallel. In MIMD systems, each processor fetches its own instructions and operates on its own data, and generally targets task-level parallelism. MIMD systems are more expensive than SIMD in terms of memory and control logic, on the contrary MIMD architectures are more general and widely applicable to larger classes of problems.

Depending on memory organization, MIMD architectures can be further subdivided into two main sub-classes: tightly coupled (or shared memory) and loosely coupled (or distributed memory) systems. In the category of tightly coupled MIMD there are two sub categories which are SMP (Symmetric MultiProcessors) and NUMA (Non-Uniform Memory Access) systems characterized by non-uniform memory access latency and higher memory bandwidth. Loosely coupled MIMD are essentially distributed memory systems like for example Beowulf clusters [302].

Many processors are nowadays “hybrids” using multiple classes of parallelism. For example, Multimedia Extension (MMX) instructions were added by Intel to the x86

SISD microarchitecture, then Streaming SIMD Extensions (SSE) and more recently Advanced Vector Extensions (AVX) instructions (supporting the simultaneous execution of four double precision or eight single precision floating-point numbers), have been added to successive generations of Intel processors including MIMD multi-cores.

Another hybrid architecture is the modern Graphics Processing Unit (GPU) that is a MIMD+SIMD architecture. A GPU contains a collection of multithreaded SIMD processors (also called SIMT processors – *Single Instruction Multiple Thread*s) each one executing the same instruction on different data streams. For example, the NVIDIA Pascal architecture [258] has 56 multithreaded SIMD processors (*Stream Multiprocessor*s) each one incorporating 64 single-precision NVidia cores for a total of 3854 cores. The interest in GPU computing blossomed when its massively parallel potential was combined with a programming language that made GPUs easier to program and to use.

In the following, we provide a brief overview of some different microarchitectures that are nowadays used in HPC and cloud environments as well as in standalone parallel computers. We consider GPUs, multi/many-cores and also distributed systems possibly aggregating all different forms of parallelism.

2.3.1 General-Purpose GPUs Architecture

One of the big differences between CPU and GPU architectures is how they use the chip area. A CPU uses most of its chip area for implementing multilevel caches to reduce the long latency to off-chip memory. Instead, GPUs rely mainly on hardware multithreading to hide the memory access cost to high-latency memories. The GPU devotes much more chip space for pure floating-point operations. Besides, GPU memory is more oriented toward providing high-bandwidth rather than low-latency and the memory capacity is smaller if compared to conventional general-purpose servers. This makes GPUs very effective for data-parallel problems with regular memory accesses and a relatively high ratio between computation and memory transfer time. Since GPUs are coprocessors, typically attached to PCIe buses, one of the main source of overhead when using GPUs is the time spent transferring data between

CPU memory to/from GPU memory. Hiding and/or reducing such transfer time is of foremost importance to benefit of the computing power of the many GPU cores. In GPU computing, performance comes from using a large number of GPU threads, which are quite light-weight entities with zero context-switching overhead. This is another important difference with respect to CPU systems whose threads are more coarse-grained entities and are usually quite lower in numbers.

GPU Computing took off when CUDA and AMD FireStream was proposed around 2006 for programming GPU cards. These programming interfaces and languages were designed by the GPU vendors to make a step forward to a usable, scalable and manageable programming model for GPU architectures. Later, the Open Compute Language (OpenCL) has been proposed to provide a unified API for heterogeneous computing on several different kinds of parallel devices, including GPUs, multi-core CPUs and more recently FPGAs [101].

Although GPUs were initially conceived for computer graphics, some pioneer efforts have been made to develop user-friendly, C-like programming languages so to write parallel programs running on graphic cards. Examples are *Cg* [232] and OpenGL Shading Language [206]. Later in 2006 NVIDIA introduced the GeForce 8800 graphic card featuring the first unified graphics and computing GPU architecture programmable in C with the *Compute Unified Device Architecture* (CUDA) parallel model. Since its first public release, CUDA has evolved to a more complete programming language that looks like a C++ program, albeit with some important restrictions. The scientific research community and software developers have then become interested in harnessing the computing power of GPU cards for general-purpose computing. This field of research is known as GPGPU ("General-Purpose computing on the GPU") [265].

In the following, we briefly recap CUDA and OpenCL programming models.

CUDA

In CUDA, the program consists of both host and device code [261]. Host code is compiled by a standard C/C++ compiler (e.g., GNU `gcc`) whereas device code is compiled by the CUDA compiler (NVIDIA `nvcc`) to an assembly language called PTX (Parallel Thread Execution) providing a stable instruction set for general purpose parallel thread execution [259].

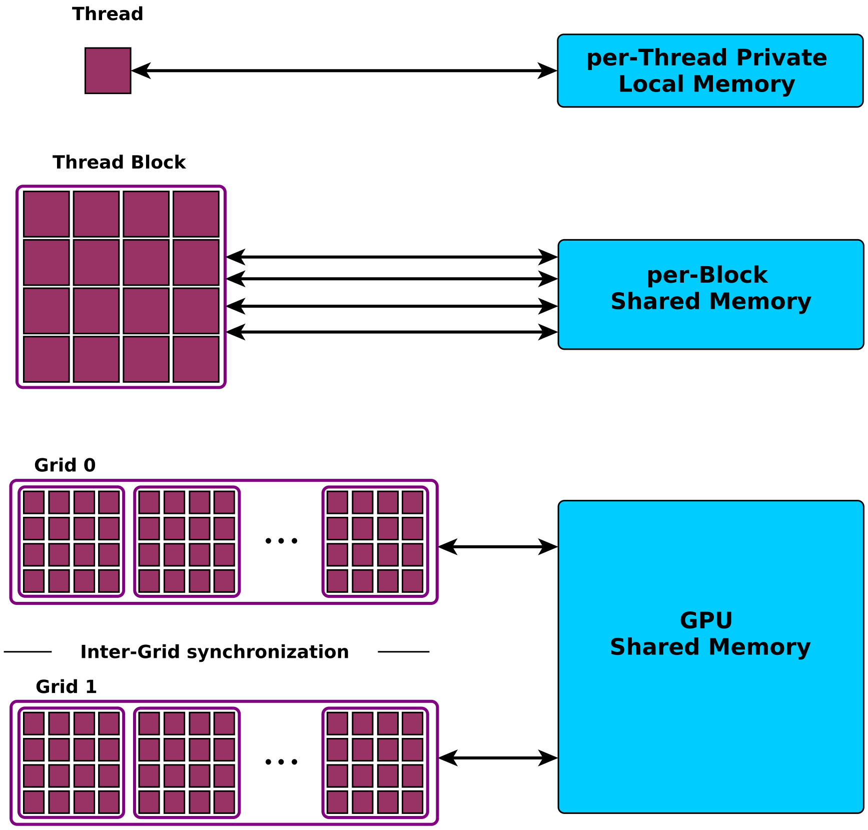


Figure 2-5: CUDA hierarchy of threads, blocks, and grids and its memory organization.

A CUDA program execution starts on a CPU where host threads transfer input data to GPU device memory before invoking the GPU kernel code. When the kernel execution finishes, output data is transferred back from the GPU device memory to the CPU main memory.

Threads in a CUDA kernel are organized in grids of blocks where each thread block internally contains multiple threads (see Figure 2-5). The maximum number of threads inside a single thread block depends on the compute capability of a GPU. Blocks of thread can be executed by a single compute unit called *Streaming Multiprocessor*. The group of threads executed together is called *warp* by Nvidia. All threads grouped into a warp perform the same instructions in a lockstep manner. It is possible that two or more threads follow different execution paths. In this case, all threads sharing a common execution path execute together while the other threads are paused. Therefore, it is beneficial to avoid warps with divergent execution paths, as this reduces the amount of threads pausing their execution. Nvidia calls this execution model: Single-Instruction, Multiple-Thread (SIMT).

GPU shared memory (also called global memory) is the largest memory but it has high latency compared to other kinds of GPU memories. The GPU can optimize accesses to the global memory when it is accessed in certain fixed patterns. Specifically, multiple accesses to the the global memory from different threads in a block can be coalesced into a single larger memory access. There are several different access patterns which are detected by the hardware and coalesced. Such optimized accesses allow us to fully exploit the available memory bandwidth. Besides global memory, each Streaming Multiprocessor has an on-chip shared memory having low latency and high bandwidth. If the programmer wants to exploit this memory, he/she has to move data between the global memory and the shared memory using explicit CUDA calls. All threads executing on the same Multiprocessor have shared access to the shared memory. Shared memory can also be used by a thread to efficiently synchronize and communicate with other threads running on the same SM.

Finally, starting from CUDA 6 one extra layer of APIs called Unified Memory for CPU/GPU memory management has been introduced. Data is allocated at the host side and migrated in a user-transparent fashion to and from the GPU memory. This new feature aims at simplify the complexities of memory management while still maintaining acceptable overall performance [217].

OpenCL

OpenCL (Open Computing Language) is a multi-company initiative promoted by the Khronos group[1] to develop a portable programming language that provides a unified computing platform for heterogeneous multi-/many-core systems. The main technological players such as Intel, Apple, NVIDIA, AMD, and ARM are part of the

[1] Khronos group web site: `https://wwww.khronos.org`

Khronos consortium and these players provided OpenCL implementations for different architectures, from standard multi-cores to GPUs, FPGAs and general purpose many-cores such as the Intel Xeon Phi. For example, the OpenCL implementation by NVIDIA runs on all NVIDIA GPUs that support the CUDA architecture. The OpenCL standard version 1.1 was ratified in 2010 (the version 1.0 was released in 2009) and it is currently the most commonly supported version of the standard [180]. OpenCL abstracts different hardware architectures providing a platform model and a common parallel programming interface.

The OpenCL platform model distinguishes between a host and multiple OpenCL devices to which the host is connected. An OpenCL application executes sequentially on the host and offloads parallel computations to the devices through a *command queue*. The host submits commands into a command queue which by-default processes all commands in the FIFO order even though it is also possible to configure command queues to operate out-of-order. An OpenCL device holds *compute units* (CU) which further may include one or more SIMD *processing elements* (PE). For a standard multi-core, the single core corresponds to a CU whereas the vector units inside a core are the PEs. In the case of GPU devices, a streaming multiprocessor is the CU and the GPU cores are PEs.

Conceptually, the OpenCL programming style is very similar to CUDA when NVIDIA GPUs are considered. The main difference between CUDA and OpenCL models is that OpenCL code is compiled dynamically by invoking specific OpenCL functions. The dynamic compilation enables better utilization of the underlying device and latest software and hardware features such as SIMD computing capability of the hardware. At the first invocation, the OpenCL code is automatically uploaded to the OpenCL device memory.

The OpenCL memory model defines four types of memories: *global*, *constant*, *local* and *private* (see Figure 2-6). Each device has its memory that is logically distinct from the host memory. Data has to be explicitly moved between the host and device memory issuing specific commands into the command queue for copying data from the host to the device and vice versa. The global memory area is shared by all CUs of

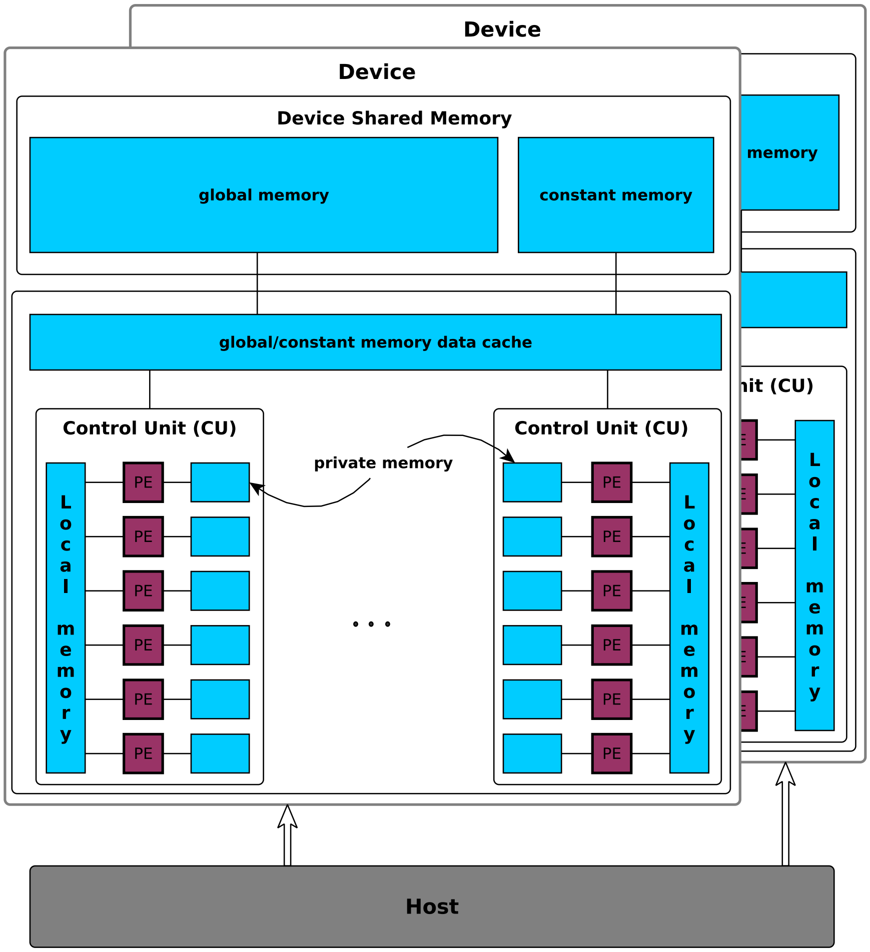


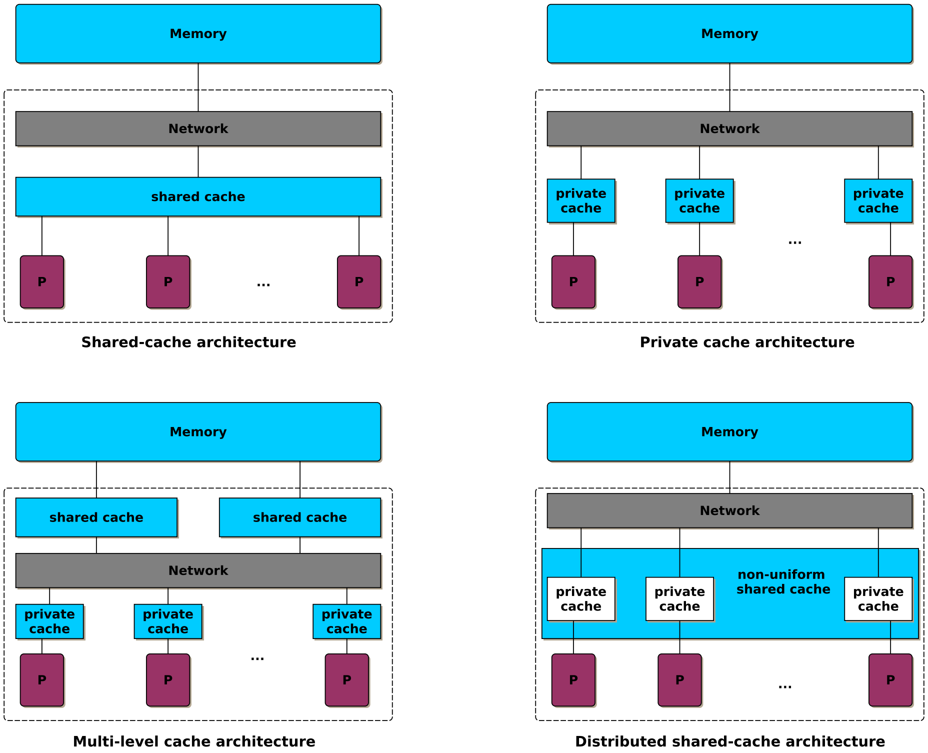
Figure 2-6: OpenCL schematic architecture.

the device and is accessible from any PE and from the host in read/write mode. As in CUDA, the global memory is the largest and slowest memory area on an OpenCL device. The constant memory is a region of the global memory which remains constant during device execution. It is initialized once by the host and remains constant with read-only access from the device. The local memory is a memory region private to a CU. It permits read and write accesses from the device; the host has no access to such memory area. The private memory is a memory region private to a processing element. Differently, from the host, the device has full read and write access to it.

The basic execution model of OpenCL relies on the concepts of *kernel* and *program*. A kernel is the basic unit of executable code applied over a data set (for data parallel execution) or as one instance of a function, to model a task parallel execution. A program is a collection of kernels and OpenCL functions for managing kernels. The execution starts on the host program, which can manage one or more OpenCL devices, by enqueuing the kernel-execution and memory-transfer commands to the command queues of the selected devices. When launching a kernel on a device, the programmer explicitly specifies how many threads will execute the kernel on the device. A thread executing a kernel is called a *work-item*. Work-items are grouped in *work-groups* which enable a more coarse-grained organization of the execution. All work-items of a work-group share the same local memory, and they may synchronize one each other because OpenCL guarantees that work-group are executed on the same CU. Synchronization between work-items of different work-groups is not allowed. Local memory can be used to make variables shared for a work-group as all work-items of the workgroup can read/write to it. Private memory is only visible to individual work-items and each work-item can modify or read only its data.

OpenCL is emerging as the *de facto* standard for programming heterogeneous systems. In principle, code written in OpenCL can be ported to all OpenCL platforms without any modification. However, device-specific optimization applied to an OpenCL code, for example, to optimize GPU execution, may negatively affect performance when moving the code to a different device (e.g., CPU device). Moreover, the OpenCL programming interface requires explicit management of many low-level details (memory transfers, kernels compilation, and workload scheduling) which require a deep understanding of all components of the system architecture. This makes OpenCL an excellent candidate to be used as a run-time system of higher-level programming abstractions relying on OpenCL for portable program execution on heterogeneous systems [27].

In this respect, an example of a parallel framework using OpenCL run-time is *SYCL* [285]. SYCL is an open industry standard for programming a heterogeneous system allowing to run standard C++ source code on either an OpenCL device or on the host. Kernels are expressed through C++ lambda functions. A SYCL program is valid C++ program that can be compiled by any C++ compliant compiler, and can be executed on the host as fallback for platforms with no OpenCL support.



Shared-cache architecture

Private cache architecture

Multi-level cache architecture

Distributed shared-cache architecture

Figure 2-7: Example of cache architectures in CMPs.

2.3.2 Multi-core and Many-core Processor Architecture

Multi-cores are tightly-coupled MIMD architectures. They are shared-memory multiprocessors systems (SMPs) integrated into a single chip, also referred to as Chip Multi-Processors (CMPs). In these systems, the physically shared memory is the primary means of cooperation among threads and processes running on different cores. Communications occur implicitly through loads and stores coordinated by synchronization protocols typically implemented using locks.

Today, almost all multi-cores implement a form of Simultaneous Multi-Threading (SMT) in which instructions are fetched from different control flows at every clock cycle. SMT technology has the potential of significantly enhancing processor computational capabilities by exploiting thread-level parallelism (TLP) inside a single physical core. An SMT core is a logical core having a private hardware context and sharing a set of functional units with other logical cores of the same physical core. This architecture reduces context switching and provides an effective mechanism for hiding high-latency accesses to off-chip memories.

All multi-core processors maintain a cache-coherent memory system of both shared and private data. The cache coherence protocol allows fast access to commonly used data in their own private caches while maintaining consistency when some other processor updates shared variables. A caching system is said to be coherent if all processors, at any point in time, have a consistent view of what is the last globally written value to each location. The most common implementation of cache coherency uses an invalidation protocol where local copies are invalidated if a thread updates a shared variable present in another cache. Various cache organization architectures have been proposed, relying on private, shared or mixed, flat or hierarchical cache structures [266]. Figure 2-7 gives a schematic overview of some cache architectures available in current CMPs.

Many-core processors are CMP systems that are designed to employ a high degree of parallelism (currently up to one thousand cores), containing a large number of simpler cores than those of general-purpose multi-cores. Cores are interconnected with sophisticated on-chip network and distributed caching system. Since they usually have the form factor of PCI cards, they are often referred to as hardware accelerators like other accelerators such as GPUs and FPGAs. One of the main differences between multi-cores and many-cores is the organization of the caching subsystem and the computing power of the single core. In multi-cores, cache coherence is automatically enforced by a sophisticated coherency protocol (e.g., MESIF or MOESI). Moreover, the single core of a multi-core system has quite complex hardware logic for out-of-order and speculative code execution. In many-core systems instead, given the higher number of cores, they have reduced cache coherency capabilities, and most of all, their cores are much less performant. In these systems, top performance is obtained by fully exploiting all cores rather than the high clock rate and large caches of each core.

A multi-core server equipped with one or more hardware accelerators (e.g., GPUs, many-cores, and FPGAs) is an example of a heterogeneous multi-/many-core architecture.

General-purpose multi-core technology has become pervasive in all fields of computing. Even in the realm of Digital Signal Processing (DSP) where, in the past, a single general-purpose control core marshaled many special purpose Application-Specific Integrated Circuits (ASICs) as part of a System on Chip (SOC) architecture. This technology shift is primarily due to the variety and the increased complexity of applications which require general-purpose processors with parallelism capabilities. Examples of these applications are software-defined radio [308] and mobile phone processors [296] that need to support several complex codecs and many different applications.

As discussed, the cache-coherent memory subsystem is what mainly characterize CMP system architectures. From the programmer perspective, automatic cache coherence introduces many benefits but also new challenges and some issues. Regarding challenges, the ability to fully exploit cache accesses often requires to define new cache-oblivious algorithms [161]. Besides, maintaining automatic coherence through cache line invalidation introduces the problem of *false-sharing* which may have a significant impact on the program performance [315]. False-sharing is a subtle source of cache misses, which arises from the use of an invalidation-based coherence protocol working at the granularity of block of data (on multi-cores, the block is the cache line storing multiple memory words – typically 8 64bit wide memory words).

Another important aspect that affects not only performance but also programmability and portability in CMP systems is the *memory consistency model* [299, 1]. A memory consistency model defines the shared-memory behavior in terms of loads and stores dealing with the ordering of operations to multiple locations with respect to all processors. The Sequential Consistency model (SC) defines the most basic consistency model. Writes to variables by different processors have to be seen in the same order by all processors. As defined by Leslie Lamport [216], a shared-memory system is Sequentially Consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." In a nutshell, the execution order within a single thread is the same

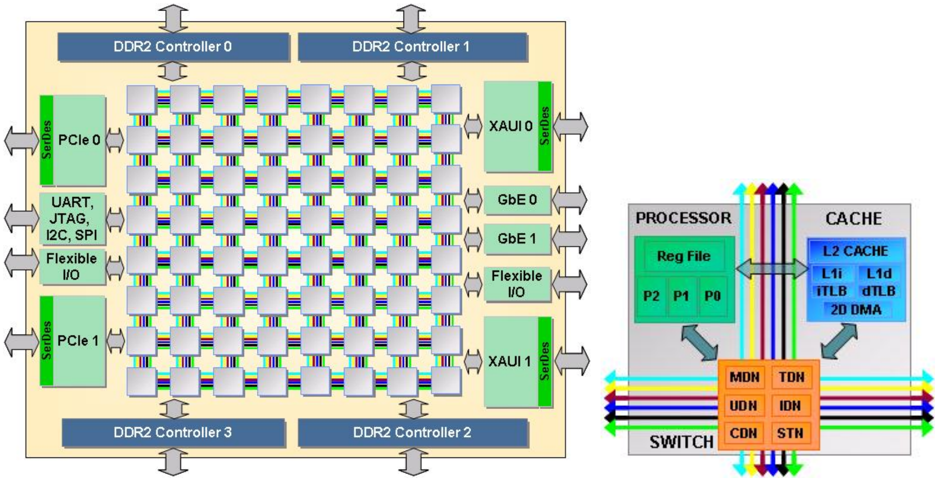


Figure 2-8: Tilera Tile*Pro*64 processor diagram (left) and schema of one of its core named *tile* (right). Figure from Wikipedia (`https://en.wikipedia.org/wiki/TILEPro64`)

as the program order, while the execution order of program between threads is not defined by SC.

Sequential Consistency prevents many common compiler and hardware optimizations as for example executing instructions out-of-order with the goal of using more Instruction Level Parallelism. To overcome the limitations of the SC model several relaxed memory models have been proposed [1]. Among these models, the *Total Store Order* (TSO) is the one used in x86-based architectures. It allows reordering of write-to-read operations while read-to-read, read-to-write, and write-to-write operations are executed in program order.

In the following, we briefly describe the architectural features of three CMP systems that are used in today HPC scenarios. Specifically, we consider two many-core architectures (Tilera Tile*Pro*64 and Intel Xeon Phi), and one multi-core architecture (Intel Xeon Skylake microarchitecture).

Tilera Tile*Pro*64. In 2008 Tilera proposed the Tile*Pro*64 multi-core processor (see Figure 2-8). It was one of the first multi-core with a high number of cores (64 identical cores – called *tiles*) interconnected with Tilera's iMesh on-chip network. The TILEPro64 is attached to a PCIe bus of a server machine and it is equipped with on-chip PCIe and network controllers. Some of the 64 tiles are not available to user programs, as they are used to run OS specific code to drive external interfaces.

Each tile is a 3-way superscalar VLIW processor running at 866MHz with a cache subsystem composed of 16KB L1i, 8KB L1d and 64 KiB L2. The cache subsystem also contains a Translation Lookaside Buffer (TLB) and a DMA engine that support automatic memory-to-cache and cache-to-cache data transfers. Differently from off-the-shelf multi-cores, no automatic memory prefetching mechanisms are provided. To sustain the memory bandwidth requirements relative to the 64 cores, the Tile*Pro*64 provides four on-chip DDR2 memory controllers that are placed at the edges of the chip. Therefore, the memory latency from a given tile depends on the tile position in the mesh and on the memory controller selected.

Virtual memory pages are "striped" among the physical controllers in 8KB chunks. The memory requests are therefore automatically spread across the different memory controllers. By using a different operating mode, the programmers may directly need to allocate entire VM pages on particular memory controllers, as usual in NUMA architectures.

Tilera also provides advanced cache management mechanisms and policies based on the concept of *Home-Node*, i.e. a tile is elected to be the home of a given cache line. This Home Node is responsible for handling the coherency relative to the line, by always maintaining an updated version and sending proper invalidations when needed. This "homing" mechanism is handled by the L2 cache of the tile, so that any local L2 space is contended by the core on the tile and the DDC. The programmer may define a specific tile as a home node for the entire memory page. If properly selected, the home node will match the core that is extensively using that page, reducing the L2 contention effect. In addition, cache-coherency can be completely disabled by defining no home node for the page, meaning that the programme must explicitly manage data coherency.

The Tile*Pro*64 processor architecture defines a relaxed consistency memory model: both load and store can be reordered (in contrast to x86 architectures where total store ordering is maintained) [299]. A memory *fence* instruction is provided to force ordering when needed.

The iMesh Network on Chip (NoC) is composed of five independent 2D meshes, each carrying a different kind of traffic. One called Memory Dynamic Network (MDN) is dedicated to memory transfers in such a way memory accesses are not influenced by other traffic, another one called User Dynamic Network (UDN) is reserved to user-level traffic. The other meshes are dedicated to I/O transfers and cache coherence protocol. The UDN mesh can be used for ultra-low-latency inter-tile message passing communications [67, 68].

Intel Xeon Phi. In 2010 Intel introduced a new technology based on the Many Integrated Core (MIC) Architecture. The first commercial version was a 22nm Knights Corner chip, released under the code-named Xeon Phi Knights Corner (KNC) [202]. The Xeon Phi coprocessor is a CMP system that can be plugged into a server host via a PCIe slot. It can be used as a coprocessor by offloading the execution of parts of the programs into the card and receiving back the results in a way similar to what happens for a GPU system. In addition, it can be used in a “native” mode, i.e. the program is loaded and executed directly in the coprocessors and can run in parallel with a program executed on the host. The two programs can communicate through an API. The Xeon Phi KNC cannot operate on its own and needs a host to provide storage and IO services. Its cores are based on the x86 microarchitecture with an in-order code execution model to reduce complexity and power consumption. The number of cores ranges from 57 to 61 (depending on the model) and the cores has a clock of about 1GHz. The Xeon Phi utilizes Simultaneous Multi-Threading on each core as a primary mechanism to hide high memory access latencies typical of in-order microarchitecture. It offers four hardware threads per core with sufficient memory and floating-point capabilities. Every core has 512 bit wide SIMD vector registers in addition to the standard x86 registers. The interconnection among cores is a ring network. Cores are connected by a high-speed bidirectional ring that allows the L2 caches of each core to be accessible by all cores for a total cache size of about 30MB. The Xeon Phi hosts from 6 to 16 GB of onboard GDDR5 RAM providing a maximum

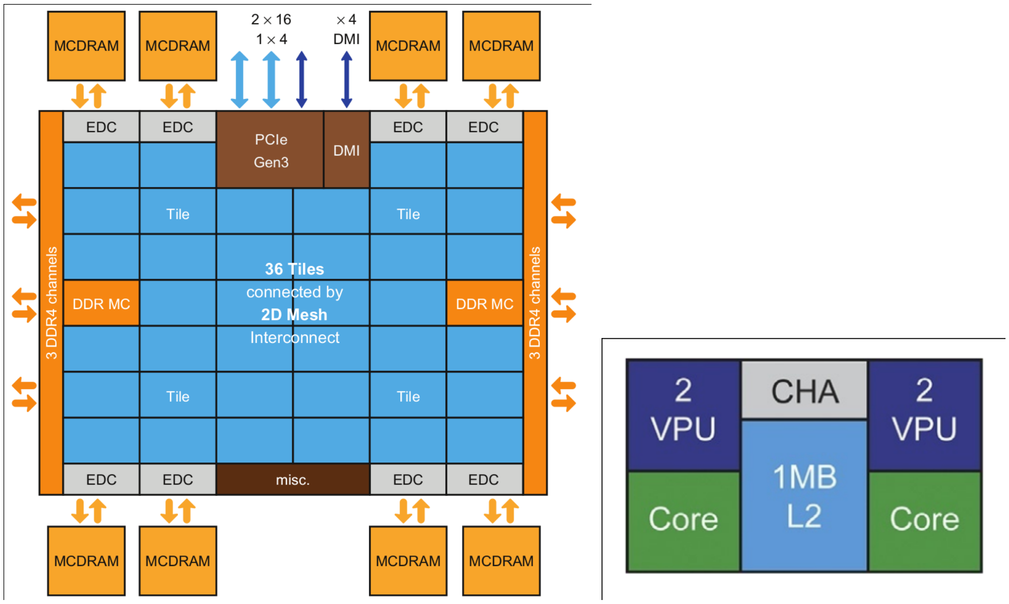


Figure 2-9: Intel Xeon Phi Knight Landing processor diagram (left) and schema of one of its tile (right). Figure from "Intel Xeon Phi Processor High Performance Programming – Knights Landing Edition" [203]

bandwidth of about 170GB/s. Each core has 32KB L1 cache that is accessible only locally.

The second generation of the MIC architecture is based on a 14 nm Knights Landing (KNL) chip featuring up to 72 physical cores and a total of 288 logical cores. Differently from the first-generation, the KNL processor is also shipped as a standalone architecture that can boot an off-the-shelf operating system offering full x86 compatibility [203]. The KNL architecture represents a performance jump compared with the previous KNC version, with a renewed on-chip interconnection network, memory sub-system with empowered performance for scalar and vector instructions and offering higher floating-point performance than previous generations.

The design on the KNL chip is separated into several tiles. It has 36 active tiles (see Figure 2-9), each one comprising 2 cores, 2 AVX-512 Vector Processing Units (VPU) and 1MB L2 Cache shared between the two cores. The core internal architecture is derived from the Intel Atom core: it is a two-wide out-of-order core with 4 SMT thread contexts and 32KB private cache.

Tiles, memory controllers, I/O controllers and other chip components are interconnected through a 2D mesh. The mesh supports the MESIF cache coherent protocol. The KNL supports two levels of memory: multi-channel 3D-stacked DRAM (MCDRAM) and double rate memory (DDR4). The former is a 16GB on-die high bandwidth memory (up to 400GB/s bandwidth), composed by 8 modules of 2 GB each, integrated on the same package and directly connected to the processor. Moreover, the chip has 2 DDR4 memory controller that allows the support of up to 384 GB RAM through 6 channels toward the external host memory.

Intel Xeon Skylake. The Skylake – the code name for the 6th-generation Intel Core microarchitecture – is a further development of the Haswell and Broadwell microarchitecture design providing many additional features and increased performance. These features include increased cores count, larger L2 cache, increased memory bandwidth, non-inclusive cache, Advanced Vector Extensions 512 (AVX-512), Memory Protection Extensions (MPX), Ultra Path Interconnect (UPI), and sub-NUMA clusters [200]. The new features of the Skylake microarchitecture are reported in Figure 2-10. The Skylake uses 14nm technology. The processors from Skylake family are scalable from a two-socket configuration up to an eight-socket configuration. The flagship processor (Platinum series) provides up to 28 cores at 2.5GHz with a CPU Thermal Design Power (TDP) ranging from 45W to 205W and with an L3 cache size of 38.5MB. As the number of CPU cores is increasing with each generation, the ring interconnection architecture utilized in previous generations (formerly Haswell and Broadwell) has been substituted by a mesh architecture to mitigate the increased latencies and bandwidth constraints associated with previous ring-based architecture. The mesh interconnection encompasses an array of vertical and horizontal bi-directional communication paths between cores, caches and I/O controller allowing also traversal from one core to another through the shortest path. The horizontal data traversal requires more cycle than the vertical data traversal. It requires one hop/cycle to move data vertically to the next core's cache, but moving horizontally, for instance between the second to the third column of cores, requires three cycles (see Figure 2-11).

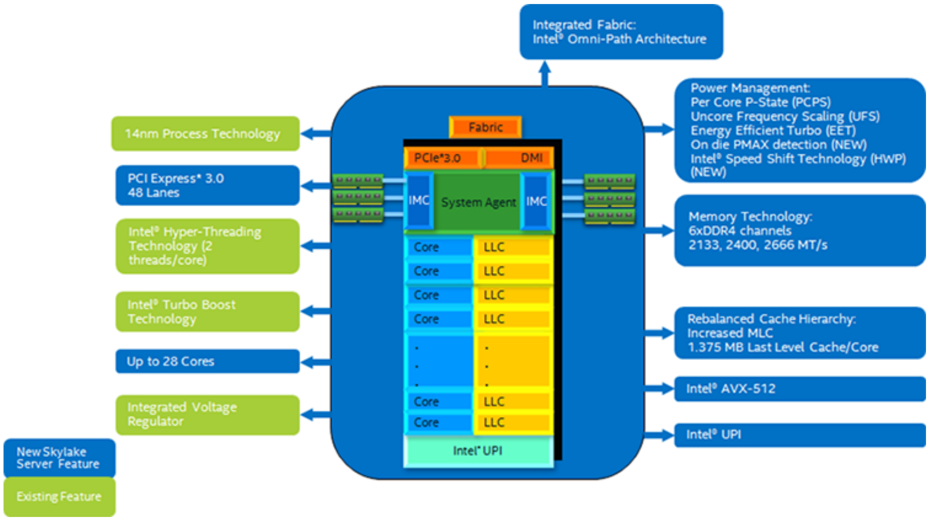

Integrated Fabric:
Intel® Omni-Path Architecture
14nm Process Technology
PCI Express* 3.0
48 Lanes
Intel® Hyper-Threading Technology (2 threads/core)
Intel® Turbo Boost Technology
Up to 28 Cores
Integrated Voltage Regulator
New Skylake Server Feature
Existing Feature
Fabric
PCIe*3.0
DMI
IMC
System Agent
IMC
Core
LLC
Intel* UPI
Power Management:
Per Core P-State (PCPS)
Uncore Frequency Scaling (UFS)
Energy Efficient Turbo (EET)
On die PMAX detection (NEW)
Intel® Speed Shift Technology (HWP) (NEW)
Memory Technology:
6xDDR4 channels
2133, 2400, 2666 MT/s
Rebalanced Cache Hierarchy:
Increased MLC
1.375 MB Last Level Cache/Core
Intel® AVX-512
Intel® UPI

Figure 2-10: Intel Skylake family processor microarchitecture. Figure from "Intel Xeon Processor Scal able Family Technical Overview" [200]

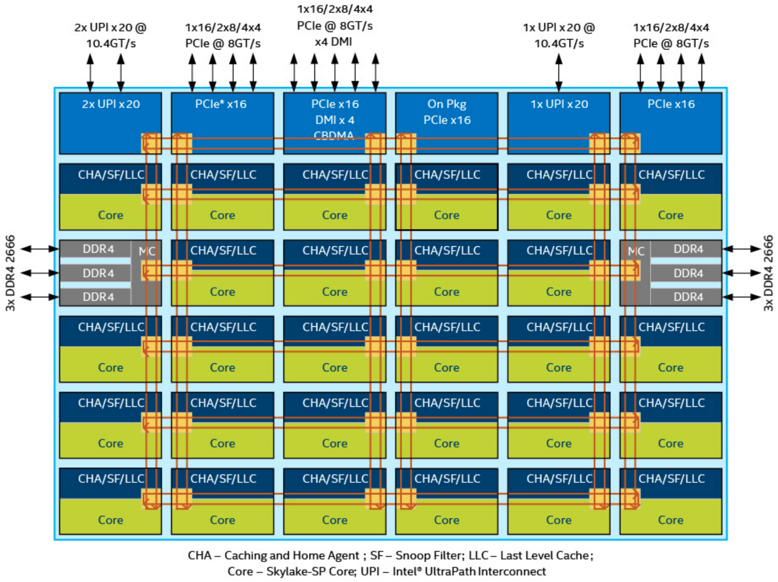

2x UPI x20 @ 10.4GT/s
1x16/2x8/4x4 PCIe @ 8GT/s
1x16/2x8/4x4 PCIe @ 8GT/s x4 DMI
1x UPI x20 @ 10.4GT/s
1x16/2x8/4x4 PCIe @ 8GT/s
2x UPI x20
PCIe* x16
PCIe x16 DMI x 4 CBDMA
On Pkg PCIe x16
1x UPI x20
PCIe x16
CHA/SF/LLC
Core
DDR4
MC
3x DDR4 2666
3x DDR4 2666
CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache;
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect

Figure 2-11: Skylake 6x6 mesh topology. Figure from WikiChip (`https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`)

The Quick Path Interconnect (QPI) that was a successful enhancement compared to the Front-Side Bus (FSB) architecture, has been replaced in the Skylake family with the Ultra Path Interconnect (UPI). The UPI is a coherent interconnect for scalable systems containing multiple processors in a single shared address space. There are up to three UPI links for connecting to other processors (3 links are used in the 8-way sockets configuration). They use directory-based home snoop coherency protocol. As

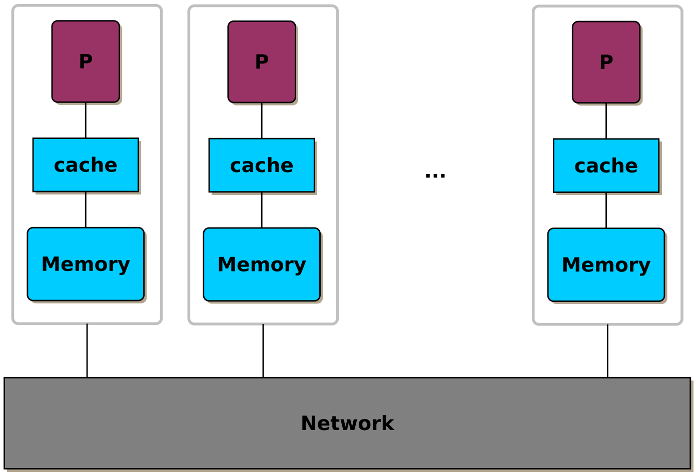


Figure 2-12: Organization of a distributed memory multiprocessor with private memories.

in the QPI, UPI also features a low power state (L0p) that reduces throughput during periods of low activity to save power. The increased efficiency of UPI allows for even lower L0p power consumption. The end result is an increase from 9.6 GT/s to 10.6 GT/s without excessive power consumption.

2.3.3 Distributed Systems

Distributed memory systems (also called *multicomputers*) are a collection of independent computers, each owning its private memory, connected through an interconnection network and forming a given topology. Figure 2-12 shows the classic organization of a distributed system, which according to Flynn's classification are examples of *loosely-coupled* MIMD architecture.

The communication and coordination of parallel programs on these systems must be done through explicit messages. Therefore, the natural programming model of distributed systems is *message-passing*. In the context of HPC systems, MPI is the de facto standard API [298].

What most characterizes distributed systems is the interconnection network. There have been several attempts by different vendors and research centers to build large-scale computers based on high-performance message-passing networks offering both low-latency and high-bandwidth communications such as InfiniBand [273], Myrinet [54] and QSNet [272].

However, outside the HPC niche, mainly dedicated to (high-cost) high-performance scientific domain, reference networks are based on standard Ethernet protocols. Today, many datacenters consist of many low-cost servers with ample data storage connected through high-bandwidth Ethernet networks (i.e. 10Gbit/40Gbit).

As a result, computing clusters have become the most widespread example of a distributed system. A cluster is a collection of computers connected over standard network interfaces and switches to form an abstraction of a single system. Clusters are generally defined as homogeneous distributed systems meaning that each cluster node is identical to other nodes. Given the separate memories, each node of a cluster runs a distinct copy of the operating system. Internally, each node is a standard shared memory multiprocessor with several cores and large caches. Cluster software is a layer that runs on top of the local OSs offering a unified abstraction, for example, for accessing the file system (e.g., DFS NFS) or for executing jobs (e.g., SLURM [328]).

Loosely-coupled distributed systems are easier to design compared to tightly-coupled distributed shared-memory systems. System architectures such as Cache-Only Memory Architecture (COMA) or Cache-Coherent-NUMA are very complex to design and faced scalability problems with limited diffusion [300].

From the programmer standpoint, employing a distributed system requires more work. It is harder to port a sequential program to a multicomputer since every communication among different parts of the program must be identified in advance and then coded using a proper communication interface. On the other hand, since communications are explicit, there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers. To simplify programming and increase sequential code portability, several attempts were made to build *Software-based Distributed Shared Memory* (SDSM) system providing a shared-memory abstraction on top of distributed memory architectures [225, 256].

In contrast to *tightly-coupled* system, distributed systems organizations favor *system dependability* and *fault-tolerance* as it is generally easier to replace a single node without compromising the functionality of the entire system. Moreover, the possibility to dynamically add and remove nodes enables horizontal scalability with an impact only on the interconnection network. Their lower cost, higher availability, and scalability make computing clusters attractive to Internet Services Providers that use such clusters for high-volume data processing [218] or for offering file, web and mail services to users often following a *pay-per-use* model.

Unlike homogeneous clusters, large-scale distributed systems are typically heterogeneous and they may comprise several collections of large and small clusters with different internal topologies as well as dedicated systems for storage, monitoring and for specific services. Although they may be classified as just large heterogeneous clusters, their architecture and technological issues make them a new class of systems. At such extreme scale, power distribution, networking, monitoring, and cooling are the main critical aspects. These systems, are the basis of the *Cloud Computing infrastructure*. Recently, a cloud model referred to as *Infrastructure-as-a-Service* (IaaS) became an increasingly popular paradigm for accessing computing resources. Providers offer computing resources and services (typically by means of virtual machines) upon request [46].

2.3.4 New Technological Trend

The rise of dark silicon, has stimulated a rethinking of systems design, where a simpler chip design enriched with function-specific hardware accelerators often provide a better balance between power consumption and performance [59]. It is thus foreseeable that future CMP systems will be packed with specialized hardware such as GPUs, Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs) in addition to Application-Specific Integrated Circuits (ASICs) implementing common operations. At any given moment in time, only a subset of these hardware components (i.e. those most suitable to solve the problem) would be powered on.

Early examples from the commercial roadmap are the ARM big.LITTLE heterogeneous processor hosting two different kinds of cores: the “LITTLE” processor is designed for maximum power efficiency while the “big” processor is designed to provide maximum performance. Both types of processor are coherent and share the same ISA [175]. Another example is the NVIDIA Jetson TK1 hosting on the same chip a low-power ARM processor (4-cores Cortex-A15) and an NVIDIA Kepler GPU featuring 192 cores [260]. Specialization of chip parts will come at the price of more complexity at the software level unless some technology breakthrough appears in the coming years.

The trend of integrating large cache and in general large memory on-chip is expected to continue in the forthcoming years. In fact, large caches allow saving power consumption since off-chip memory communications are avoided or at least reduced. In this respect, it is foreseeable that on-chip large and fast memory will be made available so to keep the entire working set of the application inside the chip area. How to organize and keep consistent such memory is one of the main challenges.

2.4 Parallel Programming Models

A parallel programming model is an abstraction of the parallel machine architecture [241]. For years, the term “parallel machine” was a synonym of a distributed multiprocessing system. These machines were typically very costly and affordable only by large laboratories either from academia or large industries, where expert parallel programmers were able to develop software exploiting the features of these machines using a limited set of machine-specific tools that allowed them to manually optimize the application code with the only objective to increase performance.

Beowulf clusters [302] emerged in the late ’90s as an economical approach to supercomputing thus contributing to the wide diffusion of parallel machines to a broader number of programmers. Still, only a restricted community of expert programmers faces the problem of programming such systems. The evolution of cost-effective clusters and then the advance of Grid computing infrastructures [160] where the parallel system became not only more and more heterogeneous but also dynamic, immediately posed the urgent need of defining suitable programming models to discourage (or to limit) the practice of using ad-hoc optimizations during software development without loosing program efficiency.

However, efficiency is not the only issue that parallel programmers have to deal with: writing portable and maintainable parallel code are also important points that a parallel programming model must address. The advent of multi-core processors has produced a radical shift in the parallel software community which had to face the challenge of defining programming models able to trade between *absolute performance*, *portability*, and *time-to-solution.*

2.4.1 Not only absolute performance

For many years, parallel computing was a synonym of absolute performance. With the diffusion of multi-cores, which retain many of the usage complexity of old parallel architectures, absolute performance figures were not the only parameters of interest when developing parallel applications. Programmer *productivity*, *total cost-to-solution* and *performance portability* may be even more important metrics. Today, multi-core systems are accepted in almost all industry segments, and from an economic standpoint, industries cannot afford the cost of re-writing and re-tuning an application for every new technological advance in computing platforms.

If on the one hand, the capability of entirely using the hardware-specific features based on a deep understanding of the underlying micro-architecture remains crucial for extracting the highest levels of performance and energy efficiency [211, 289, 203], on the other hand, the shift to heterogeneous multi-cores and the rapid architectural evolution has dramatically increased the complexity of software development that cannot be handled by using low-level tools.

We do believe that parallel programmers should focus on "the 3P" criteria (*Programmability*, *Portability*, and *Performance*) in a synergistic manner without privileging one specific "P". Therefore, the 3P criteria should be the primary metric when developing and evaluating parallel applications[2].

- *Programmability:* Programmability is related to development costs, both for what concerns the time spent to reach the final solution and programmer effort

[2] Sarita Adve first introduced the 3P criteria for evaluating memory consistency models [2].

in writing the code. One of the ambitions of programmability in the context of parallel programming is to remain as close as possible to traditional sequential programming. Parallel development overheads are due to the direct management of synchronization, communication, and task scheduling. Proper handling of these essential aspects requires to introduce a significant amount of extra-code that is needed only for the management of parallelism and concurrency.

Programmability is often paired with *code reuse* that is the capability of reusing existing well optimized sequential code. Such sequential codes are often the result of several years of development, debugging and fine-tuning. Rewriting them is therefore merely an unfeasible hypothesis. Consequently, any parallel programming framework must support the re-use of existing code as much as possible. This also means that it must offer the possibility to link and orchestrate parallel executions of different instances of such existing sequential code.

High-level parallel programming based on *parallel patterns* and *skeletons* tries to address the programmability challenge by clearly decoupling non-functional code from the business logic code promoting existing code reuse. Regarding programmability, high-level models widen the spectrum of possibility and narrow the distance with what is currently done in sequential programming.

- *Portability*: Functional portability is usually ensured by using standard libraries and tools. Such libraries guarantee that the same software can be recompiled and executed on different platforms. For example, an MPI program running on a supercomputer can be quickly ported on a single workstation without spending too much effort. Conversely, there is no guarantee on the level of performance the application programmer can expect. Such MPI programs may run even slower than the functionally-equivalent sequential program on the selected workstation. Therefore, the real problem is not functional portability (though it is an important aspect) instead is *performance portability* [270, 242]. *Performance portability is the ability to compile the same code on different plat-*

forms maintaining the desired level of performance considering the features and the peak-performance of the target platform. Recently a definition of *performance portability* has been proposed as “A measurement of an applications performance efficiency for a given problem that can be executed correctly on all platforms in a given set.”, along with the metric to quantify it by using the harmonic mean [269]. The equation proposed states that, on a given set of platforms H, the *performance portability* $PP(a, p, H)$, of an application a running a problem p is the harmonic mean of the performance efficiencies $e_i(a, p)$ on each platform i. $PP(a, p, H)$ is 0, if any platform in H is not supported by a running the problem p. However, this quantitative definition is not widely accepted yet. One of the reasons is that it does not penalize some huge development efforts for a given target platform.

A parallel application should exhibit enough hardware abstraction to hide low-level hardware architecture details, accommodate hardware heterogeneity and provide good software portability across different parallel hardware. A parallel application should also be able to adapt to new underlying architectures to maintain a reasonable level of execution efficiency. Currently, when a parallel program is ported to new parallel hardware, often it requires substantial rewriting and tuning of the code to maintain the necessary level of performance.

Owing to its high-level nature, a high-level programming model is suitable for meeting the performance portability requirement. However, this is not sufficient. The run-time support implementation should be able to dynamically adapt to different platform features and different environmental conditions, such as the presence of other parallel applications running on the same system. Rapid architectural evolution renders manual code adaptation unfeasible and mandates adaptive solutions [131, 128].

A different approach tackling performance portability is based on Domain-Specific Languages (DSLs) which trade productivity and performance at the cost of generality at the application level [81]. These approaches, in principle more efficient than general-purpose approaches, focus on expressing highly specialized abstractions for a specific application domain that can be compiled and mapped to multiple hardware architectures. Examples of frameworks providing a platform-neutral intermediate representation for DSLs are DeLite [305] and PENCIL [41].

- *Performance*: What is in general expected from a parallel program, is that the time spent to execute it on a parallel machine with n processing elements is about $1/n$ of the time spent running the sequential program that solves the same problem in one of the available processing elements. Unfortunately, such desirable behavior is not always attainable. On the one hand, not all problems are suitable to be parallelized [176]. On the other hand, even those problems suitable to be parallelized may contain a fraction of work that cannot be parallelized. The well-known "Amdahl's Law" [29] gives a theoretical limit. The amount of non-parallelizable code in an application determines the maximum theoretical speedup it is possible to achieve. If for example, 10% of the application is sequential by nature, even infinite parallelism will not speed up the application by more than a factor of ten. However, the overhead in the parallelization further limits the possibility of achieving the theoretical maximum stated by Amdahl's law. It is easy to see that when taking into account overheads, the real limit in the speedup of a parallel application is even smaller than the one determined by the fraction of the non-parallelizable parts.

  Amdahl's Law provides a performance upper bound to programs working on a fixed data size with the only possibility to increase the computing resources. In other words, the increase in the speedup for increasing problem size cannot be captured even though newer and more performant computers usually allow solving more massive problems due to the improved capabilities. In [182], John Gustafson noted that as the problem size increases, the amount of parallel work usually grows much faster than its serial counterpart. Therefore, the serial fraction of work decreases and according to Amdahl's law the speedup increases.

Thus, as more and more computing resources are added, the same application should solve bigger problems in the same amount of time. Gustafson's law holds only if the serial part grows much more slowly than the parallel one, and most of all if also the overhead of increasing the number of computing resources scales with the problem size. There exist more complex analysis methods which study how problem size and number of processors used have to be related for obtaining a given efficiency. An example is the use of *isoefficiency* metric function [181].

Another critical performance metric is *scalability*. Scalability refers to the ability of a program to utilize increasing available resources (both memory and processing elements). Many problems, such as the emerging services for data-analytics and social media, run on data-center servers. These applications operate on massive datasets that are split across a large number of server nodes for scalability. Each node concurrently services large numbers of independent requests, and inter-node communication is restricted to task management and coordination with minimal or no data-sharing. For these cases, it is of foremost importance that the underlying run-time does not introduce unnecessary overheads that could hinder full scalability. These peculiarities distinguish *scale-out* infrastructures and deployments from *scale-up* counterparts, which in contrast exhibit more prevalent data sharing, requiring extensive coherence and communication infrastructure and are typical of high-performance scientific domains.

2.4.2 Types of parallelism

It is widely acknowledged that there are two main types of parallelism, *data parallelism* and *task parallelism*. The first model refers to the possibility of performing the same functional operation on separate pieces (or partitions) of the same data in parallel, whereas *task parallelism* refers to those parallel executions that can be organized on the basis of inter-dependencies among separate tasks composing the program. Task parallelism is often specified through *task-dependency graphs*. Once the concept of sequences of input data elements is considered, the task parallelism model can be further refined by introducing *stream parallelism*. A stream is a possibly infinite sequence of values, all of them having the same data type, e.g., a stream of images (not necessarily all having the same format), a stream of files, a stream of network packets, a stream of bits, and so on. Without loss of generality, an application can operate either on distinct values organized as *streams* of data elements or on a single (possibly large) collection of data values. This leads us to make the distinction between three main type of parallelism:

- *Data Parallelism* is a method for parallelizing a single collection of data by processing independent sub-collections of elements in parallel. The input collection of data, possibly but not necessarily coming from a stream, is split into multiple sub-collections (or partitions) each one computed in parallel by applying the same function to each partition. The results produced are collected in one single output collection, usually having the same type and size of the input (this is the case for the *Map* pattern – see Section 2.5.3). The computation on the sub-collections may be completely independent, meaning that the computation uses data only coming from the current sub-collection, or, instead may be dependent on previously computed data (in this latter case, the function applied to the sub-collection might have an internal state). Therefore, data parallelism can be characterized by replication of functions and partitioning of data. Its primary objective is to reduce the *completion time* (i.e. the total latency) of the entire computation on the initial collection. It is worth noting that, data decomposition using large partitions, together with a static assignment of such partitions to independent processing elements, may introduce *workload imbalance* during the computation due to the potential variable calculation times associated to each distinct partition. Numerous smart techniques and algorithms have been proposed to define static and dynamic assignment of partitions to processing elements, among these the *work-stealing* algorithm is one the most well-known and widely used [52].

  One of the primary sources of data parallelism are loops and in general iterative computations, where successive iterations working on independent or read-only data can be executed in parallel.

- *Stream Parallelism* is a method for parallelizing the execution of a stream of elements (or collections of elements) through a series of *sequential* or *parallel* modules [301]. Parallelism is achieved by running each module (often called *filter*) simultaneously on subsequent or independent data elements. Communication between distinct filters takes place through *communication channels* preserving data ordering. Stream parallel systems are often visualized as directed graphs with several sources and sink modules and a number of filters atomically computing stateless or stateful user's functions. A stream is a sequence of values (the sequence may be infinite), all of them having the same data type. The typical requirement of a streaming application is to guarantee a given QoS imposed by the application context. That means that the modules of the streaming network representing the application have to be able to sustain a given throughput. There are many applications in which the input streams are primitive as they are generated by external sources (e.g., HW sensors) or I/O devices. However, there are cases in which streams are not primitive, and sequences of data elements are generated directly within the program [18]. Stream parallelism can also be applied when there exists a total or partial order in a computation, preventing the use of other types of parallelism. Furthermore, data streaming reduces the impact of long memory latencies and in general, the *memory wall* problem because the producer-consumer pipeline of filters naturally hides latency and favors on-chip communications through cache memories [275]. All these aspects make stream parallelism a first-class citizen in the context of parallel computing.

  Streaming network of filters underpin the formal definition of *Kahn Process Networks* a distributed model of computation that describes a program as a set of concurrent entities communicating through unbounded FIFO channels, where read operations are blocking and write operations are non-blocking [204].

- *Task Parallelism* is a method for parallelizing the execution of a function (by decomposing it in multiple sub-functions) or a set of distinct functions by running each of them according to an application-dependency graph (also referred to *task-dependency graph*). In this class, we can also include recursive fork-join computations. Functions (or tasks) are processed concurrently by threads or processes which communicate to satisfy input data dependency as described by the dependency graph. As in data-parallel computations, task parallel ones operate on a single data collection. In the case of sequences of input elements, this model falls into the stream parallelism model.

  Task parallelism is sometimes associated with the *data-flow* model of computation [220]. The data-flow model facilitates the construction of parallel programs by leveraging a run-time scheduler that is aware of dependencies between different tasks and is able to balance the workload among a set of executors. The atomic unit being scheduled is referred to as *task*. Each task is labeled with a memory access mode, e.g., input, output or input/output dependencies. These labels determine the memory side-effects produced by the task on the arguments. The scheduler dynamically tracks dependencies between tasks and if needed can also change the execution order of tasks to improve the utilization of executors (often called Workers), while still respecting the dependencies between tasks. Firing rules define when tasks are ready to be scheduled (fireble) [20, 116].

2.4.3 Abstract Models of Parallel Computations

An abstract model of parallel computation defines an abstract parallel machine, its primary operations (e.g., process creation, send/receive, read/write) and how these operations affect the state of parallel computations. The model also defines the constraints of when and where these operations can be used, and how they can be composed to create a parallel program suitable to be executed on the abstract machine [229, 297, 207].

At first glance, abstract models may appear to be useless for developing real-world parallel applications. However, abstract machine models have been intensely utilized to study parallel and distributed algorithms and for evaluating their performance independently from the actual parallel platform that eventually will execute the code. Since abstract models do not consider some practical aspects of real parallel and distributed systems, this makes easier to focus on the computational aspects of the algorithm to find performance bounds and complexity estimates.

Of particular importance is the *cost model* associated with the abstract parallel machine, which describes how to predict the cost of executing the entire parallel program. The total cost is computed by assigning a cost to each basic operation, and it is usually measured by considering execution time units, the total number of operations needed to solve the problem or the number of used resources.

In the following, we describe some relevant abstract models.

PRAM. The Parallel Random Access Machine model (PRAM) was introduced by Fortune and Wyllie in 1978 as a model for general purpose computations [158]. It is an extension of the Random Access Machine (RAM) model used in the design and analysis of sequential algorithms.

The PRAM assumes a set of p processors connected to a single shared memory space. Each processor has a set of local registers, knows its index i, and has instructions for direct and indirect read/write access to the shared memory. There is a single global clock that feeds both processors and memory. At every cycle, each processor performs one of the following operations: i) read a global memory location; ii) write into the memory of a value stored in a local register; iii) compute an operation using values stored in its local registers.

The execution of each instruction, including memory accesses and regardless of the access pattern, takes one unit of time. Besides, there is no limit on the number of processors accessing shared memory simultaneously. The memory consistency model of the PRAM is *strict consistency*: a write at cycle c becomes visible to all processors at the beginning of cycle $c+1$. The strict consistency memory model is the strongest consistency model known [1].

The PRAM model is refined by defining the effect of multiple processors writing or reading the same memory location in the same clock cycle. In the EREW (Exclusive Read Exclusive Write) PRAM, the memory location is exclusively read or written by at most one processor, in the CREW (Concurrent Read Exclusive Write) PRAM concurrent reading are allowed, while in the CRCW (Concurrent Reading Concurrent Writing) PRAM model even simultaneous write accesses by different processor to the same memory location are allowed. When concurrent writes are allowed, the CRCW PRAM model defines different sub-models depending on how such concurrent accesses are resolved. In the *common* model more than one processor can write the same location but only with the same value; in the *arbitrary* model the processor which successfully writes its value is decided arbitrarily; in the *priority* model the winner processor is the one with higher priority (if a priority exists); finally, in the *combining* model the value that is written is a linear combination (e.g., a reduction) of all values that should be written in the same memory location.

The cost model is simple: the total time spent to execute an algorithm, by all processor, is the quantity pT_p where T_p is the time used to run the algorithm with p processors. The PRAM model completely hides aspects such as synchronizations, data locality exploitation, and data transfer. Also, it does not consider essential aspects for reliable prediction of execution costs such as communication latency and/or available bandwidth. For hiding communications cost one possibility is the exploitation of a certain amount of excess parallelism also called *parallel slackness* [316].

Many variants of the PRAM model have been proposed mainly within the parallel algorithms theory community. Such models relax one or several of the PRAM's properties. These include Asynchronous PRAM (A-PRAM) [166, 95] and the Hierarchical PRAM (H-PRAM) [192].

Asynchronous Shared Memory. An *asynchronous shared memory* model consists of a finite collection of processes (or threads) interacting with each other through a limited collection of shared variables. At first glance, this model resembles the PRAM model, but its processes execution is entirely asynchronous. Each process can interleave its operations in an arbitrary order, with no bounds on single process speeds. Conflicting accesses to shared variables must be resolved directly by the programmer by using features and mechanisms of the underlying system (e.g., atomic operations, locks or semaphores). A crucial aspect is the visibility of write operations which defines the memory consistency model. To improve efficiency with respect to the strict consistency memory model of PRAM, several weaker consistency models have been proposed such as for example the Total Store Ordering (TSO) and Weak Ordering (WO) [299, 1].

POSIX threads (PThreads) is currently the most widely used parallel library implementing the asynchronous shared memory model. It lacks a well-defined memory model, therefore, the PThreads programmer inherits the memory consistency model offered by the underlying hardware platform.

The cost model of the asynchronous shared memory model depends on the realization of the shared address space. In SMP systems without caches, the cost of accessing shared variable is the same independently of the address and the requesting processor. In a NUMA multiprocessor, the cost of the shared memory access depends on the distance of the memory location from the requesting process. In general, the access time to shared variable depends on where the shared variable is stored: in the local processor cache, in the local memory or in a remote memory. The cost associated with accessing these memory levels may vary from one to more orders of magnitude for each level. Essential parts of the cost for accessing a given memory location is the time spent for having exclusive access to the memory location and the number of requests the memory subsystem has to serve for a unit of time. Several attempts have been made to define reliable cost models for different architectures, some of them are based on queueing theory [68, 215, 318] others on concrete performance observations of modern systems such as the well-know Roofline model [324], others more recent proposals are based on hylomorphisms [79].

In the context of shared memory model, of particular relevance is the *transactional memory model* [190]. It adopts the concept of transaction known from DBMS as a primitive for parallel programming of asynchronous shared memory systems. A transaction is a sequential code section enclosed in a statement (e.g., *transaction*{}) that is guaranteed to either commit completely to shared memory or to fail. No intermediate state of a transaction is allowed to be visible to other processes. Transactional memory provides a declarative approach which leaves the implementation of atomicity to the run-time system (in this case it is referred to as Software Transactional Memory) [294] or directly to the hardware (Hardware Transactional Memory) [286].

A different approach to tackle the memory conflict problem, is the implementation of *lock-free* concurrent data structures [252] which avoids idle times resulting from using locks. Such implementations, leverage on hardware-supported atomic instructions that combine a load and a store operation such as for example *compare-and-swap* (CAS) or *load-linked/store-conditional* (LL/SC).

LogP. The LogP model [99] is an asynchronous model describing distributed memory systems which communicate by using point-to-point non-blocking primitives. A LogP program is defined as a set of asynchronously cooperating processes exchanging data with an associated cost. The estimation of the communications cost is given without considering the real topology of the interconnection network. A LogP machine comprises a set of processors each one with its local memory and connected one each other via an interconnection network. The parameters describing a LogP machine are four: 1) the latency L which specifies an upper bound on the latency of the interconnection network; 2) the overhead o which specifies the overhead associated with the transmission or reception of a message; 3) the gap g which gives the time that must pass between two successive send operations of a processor (its reciprocal is the per-processor bandwidth); 4) the number of processors P.

To design a parallel algorithm by using the LogP model requires to balance both computational and communication loads due to the limited available bandwidth. In addition, an accurate scheduling and mapping of parallel activities are needed to avoid bottlenecks during the program execution.

The LogP model has been extended to the LogGP model [26] by introducing another parameter G modeling the difference in bandwidth when sending short or long messages.

BSP. The Bulk-Synchronous Parallel (BSP) model, is a model proposed by Leslie Valiant in 1990 [316], proposing a "bridging" model between parallel software and the underlying parallel machine. As the von Neumann model provides a unifying approach bridging the two worlds of sequential hardware and software, likewise the BSP model provides a unifying model that bridges between parallel hardware and software.

A BSP computer is made by two main components: 1) a set of processor each one with its local memory; 2) a communication network. The basic assumptions are: i) accessing local memory is faster than accessing remote memory; ii) the algorithm designer should not worry about network details, but only about global application performance.

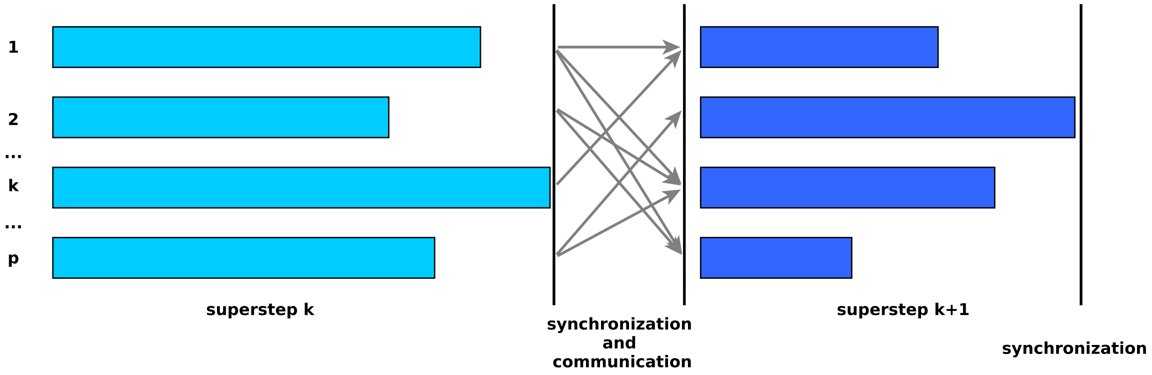


Figure 2-13: BSP supersteps.

A BSP algorithm consists of computation and communication *supersteps*. In each superstep processors compute using local variable only, then follows a global interprocessor communication phase where data are exchanged among processors. The communication takes place only at the end of a superstep when all processors execute a synchronization barrier. The messages sent or received during a superstep can be used only after the barrier has been crossed and so at the beginning of the next superstep. The general structure of a BSP algorithm is sketched in Figure 2-13.

The cost model involves three parameters: 1) the number of processors p; 2) the *permeability* g, that is the time spent to deliver a word; 3) the *periodicity* L, that is the minimal time (expressed in local computation step) between two distinct synchronizations, in other words, it is the overhead associated to the communication set-up and to the execution of the local work[3].

A *h-relation* is a superstep in which every processor sends and receives at most h words. The h-relation determines the cost of a superstep that can be computed as hg. By considering w as the maximum amount of local computation performed by each processor, the cost of a superstep is then give by: $T_{spep} = w + hg + L$. The cost of the entire program is then simply determined by summing up the costs of all supersteps executed.

The BSP model allows us to derive realistic predictions of execution time and can thereby aid the programmer to design parallel programs. The model has inspired several parallel programming interfaces in different host languages: BSPlib [194] for the C language, BSML [227] for OCaml, Apache Hama [295] for the Java language and recently Bulk [72] for the modern C++17 language.

2.4.4 Fundamental models of concurrency

Communicating Sequential Processes (CSP) [196] was proposed in 1978 by Hoare as a new language for modeling concurrent systems. The CSP model introduced a number of important concepts that can be found in many modern programming languages. Later the CSP model was used to develop the process calculus theory. In CSP, processes communicate exclusively through explicit messages without shared memory, and no other synchronization mechanism is needed. The control of input non-determinism via the *guarded command* and the *rendez-vous* of inter-process communications, are two key concepts of the CSP model. A concrete implementation of the CSP model was the Occam language used for programming Transputer microprocessors. Recently, Google proposed the Go language [141], a C++-like imperative language heavily inspired by the CSP model.

The **Actor model** is a concurrent programming model first proposed by Hewitt et al. [191] in the context of Artificial Intelligence. Later, the actor model has been

[3] It may be seen as the cost of executing an empty superstep.

formalized by Agha [4, 5]. Actors are concurrent entities, which interact exclusively via asynchronous messages. They are uniquely identified by an opaque identifier so that they can be transparently addressed during send operations. By providing network-transparent messaging, the actor model offers a high-level of abstraction for designing applications targeting parallel and distributed systems. Each actor buffers input messages in a mailbox and processes them sequentially in a single logical step thus avoiding non-determinism in actors execution. Upon receiving a message, an actor can: (1) send messages to other actors, (2) spawn new actors to distribute workload and (3) change its internal behavior to process subsequent input messages differently. Such an event-based computation model prevents blocking waits for specific messages, which helps avoiding deadlocks in complex programs. Moreover, since actors can only interact via message-passing, there is no shared state between actors so that they never corrupt each other internal state (i.e. local variables) avoiding potential race conditions. The lack of shared state among actors together with asynchronous messaging enables actor programs to exploit the processing capabilities of multi-core platforms, potentially. Also, since actors are not tied to the specific physical machine because of their opaque addressing, the run-time systems can distribute actors across multiple devices de facto enabling strong scalability.

The **Data-Flow** model is a programming paradigm modeling a parallel program as a directed graph where operations are represented by nodes while edges model data dependencies. Nodes represent the functional unit of computations that are fired when all input data items are present. Operations without direct dependences as well as operations that become fireable at the same time can be executed in parallel. Differently from the actor model, in the data-flow model, the structure of the parallel program (i.e. its concurrency graph) is fixed and all data dependencies are statically defined. A data-flow graph can be executed directly by mapping each node to a process (or thread) and allowing them to communicate via FIFO channels, alternatively in the *dynamic scheduling* approach, a scheduler tracks the availability of tokens in input to each node and executes the ones that are ready [220]. The data-flow model has been formalized by Kahn in 1974 [204].

2.4.5 Evolution of classical parallel models

In the late '90s, two approaches become predominant in the HPC parallel programming landscape: OpenMP for shared memory systems [102] and MPI for distributed memory platforms [298]. These two standards are two important representatives of the so-called *classical or pure parallel models* [138]. *Message passing* and *shared address space* represent two distinct programming models, each of them providing a well-defined paradigm for sharing, communication, and synchronization. The shared address space model (also called *shared memory model*) is a convenient programming model enabling data sharing through natural mechanisms of reading and writing data structures stored in a common address space. This is the classical model of concurrent programming used by POSIX Thread (PThreads) standard [70]. In the PThreads model, the global variables and dynamically allocated heap memory areas are shared by threads. This can cause programming difficulties, forcing the programmer to deal with thread local storage and mutexes to protect critical sections. Moreover, PThreads does not provide any parallel memory model that defines the behavior of parallel programs for memory accesses. This limitation has been overcome in the C++11 standard in 2011 [96], where multithreading programming was introduced in the C++ language with a complete parallel memory model [55].

The OpenMP programming model tried to simplify shared memory programming introducing higher-level concepts such as *parallel-for* for parallelizing loops with independent iterations and more recently task-based programming starting from version 3.0. While PThreads is implemented as a library, OpenMP is implemented as a combination of a set of compiler directives, pragmas, and a run-time providing both management of the thread pool and a set of library routines. These directives instruct the compiler to create threads, perform synchronization operations, and manage shared memory.

Recently, the PGAS paradigm (Partitioned Global Address Space) revamped the Distributed Shared Memory approach that was very popular in '90s [225] by adding syntactic mechanisms to control data locality in a parallel application executed on a distributed system. The UPC (Unified Parallel C) [145] is an example of the PGAS approach which extended the C language. In UPC, any processor can directly read and write variables on the partitioned address space, while each variable is physically associated with a single node. This programming model is still a shared-memory model that uses barriers and locks to synchronize the execution flow.

Concerning the message passing model, it is a parallel programming model where communication between processes happens by explicit messages. It is a natural model for a distributed memory system, where communication cannot be directly implemented by sharing variables. The Message Passing Interface (MPI) standard, is the reference API to construct parallel applications composed of tens to hundreds of thousands of communicating processes. Each MPI application, despite its complexity, is conceived as a single program designed having precise knowledge of data partitioning and communication patterns.

The evolution of parallel computing platforms toward heterogeneous many-core systems and clusters of multi-cores equipped with accelerators has blurred the clear boundary between the message passing and shared memory models. In the context of HPC, the mainstream programming paradigm evolved into the so-called MPI + X approach, where the X part is a model which mainly focuses on programming a single node of the MPI network, for example by using OpenMP or OpenACC [263] or CUDA [261] or OpenCL [180].

2.4.6 Parallelization approaches

There are basically two main approaches to program parallelization: *autoparallelization* and *explicit parallel programming* [138]. Autoparallelization deals with parallelization of sequential code using a compiler which can automatically detect parallelism and automatically performs data allocation and task scheduling. This is considered as the “holy grail” of compilers and parallel computing research communities since decades. This line of research has had advances in the exploitation of *Instruction Level Parallelism* (ILP) through automatic vectorization of sequential code [230]. Instead, the automatic extraction of thread-level parallelism needed for exploiting multi-core platforms had only limited success due to the need for complex program analysis during the compilation phases [74]. Examples of automatic parallelization compilers and tools are PLUTO [57] and Par4All [30]. Due to the complexity of automatically transforming sequential algorithms into parallel ones, the amount of parallelism extracted using this approach is low. This outlined the need for higher-level programming models which allow programmers to instruct compilers by introducing modifications to the original program to guide the compilation process in introducing parallelism. New programs have to be written directly in parallel, avoiding to follow the path of writing first the sequential code and then trying to parallelize it.

Nowadays almost all computing systems are hierarchical with different levels of potential parallelism. Consequently, the programming model needs to address all different forms of hardware parallelism in a manner that is abstract enough to avoid limiting the implementation to a particular kind of hardware. The way the programmers provide "hits" to the compilation phase can be more-or-less explicit and more-or-less intrusive in the program code. Over the years several initiatives propose different approaches, from compiler directives such as in the OpenMP to libraries such as Intel TBB [282] and FastFlow [21] to language extensions such as in Cilk [51] to utterly new programming languages such as X10 [86].

2.5 High-Level Parallel Programming

In sequential programming, software portability and code maintainability have been dealt with by raising the abstraction level of programming and introducing several software layers from high-level requirements specification to low-level code generation for the target platform. Porting a sequential program to a different machine, typically requires just to recompile the code or at worst to tweak a minimal number of configuration parameters to let the programming environment generate a different target program.

In the parallel computing scenario, we are still far away from this goal. Parallel programmers are still influenced by the HPC community approach so that they tend to write parallel code by using low-level mechanisms and libraries that allow retaining complete control over the underlying platform. This approach limits not only code portability but most of all *performance portability*.

As is typical in computer science, abstraction of the problem and the utilization of high-level methodologies are what provide the best answer to the given problem. In the context of parallel programming, for instance, this means that threads, synchronizations, and communications concepts have to be abstracted out in higher-level entities and constructs having a precise semantics. Intel Threading Building Blocks [282], OpenMP [102], and Cilk [51] are three examples of widely used high-level parallel frameworks that provide those kinds of abstractions each one in a different way.

The programmer builds parallel applications by using and possibly composing high-level constructs that are guaranteed to perform well on a wide range of parallel systems. These constructs hide the actual complexity of dealing with threads and synchronizations and abstract out all the details of the underlying architecture. In this way, the programmer has only an abstract high-level view of the parallel program and can concentrate only on computational aspects (i.e. functional code) leaving the most critical implementation decisions to the run-time system executing the non-functional code. By using different run-time implementations of high-level constructs, it is possible to enable not only portability of code but also performance portability as several different implementations of the same construct can be selected depending on the target platform. One of the main advantages of a high-level approach is to make programming efforts less time-consuming so to increase productivity in software development [123, 297].

Any parallel computation can be described as a data-flow graph of parallel activities where nodes represent tasks to be executed and arcs represent dependencies between nodes. Specifically, two distinct types of relations among nodes can be defined: a) *control dependency*, i.e. those dependencies explicitly given by the programmer and to establish an ordering in the execution of concurrent activities; b) *data dependency*, i.e. those dependencies on data where a data item produced by a node is needed to compute the concurrent activity by another node. The parallelism arises implicitly by executing in parallel tasks not linked by any dependency arc. Activities linked by a control or data dependency form a sub-graph whose tasks have to be executed respecting the ordering. In these cases, parallelism arises both exploiting *stream parallelism* (e.g., data-flow pipelines) and executing independent sub-graph in parallel.

A number of parallel frameworks have been proposed over the years that provide different implementations of the graph of concurrent activities by using different methods for coordinating the execution of tasks, moving data among concurrent entities and different techniques for synchronizing dependent tasks. Examples are ASSIST [317], CnC [208], StarSs [274], StreamIt [310] just to mention some of them.

One of the most widely accepted approaches for raising the level of abstraction in parallel programming is based on the concepts of *parallel patterns* [241] and *algorithmic skeletons* [94], which are schemas of parallel computations that recur in many applications and algorithms and that are made available to programmers as high-level programming constructs with a well-defined functional and non-functional semantics [15]. Each parallel pattern has one or more implementation skeletons depending on the parallel platform considered. Different models of communications synchronizations and coordination of task execution can be used to implement a given parallel pattern.

In the following, we briefly introduce structured parallel programming, focusing in particular on the skeleton-based approach.

2.5.1 Structured Parallel Programming

Parallelism exploitation is characterized by several complex problems that need to be solved in a synergistic way. For example, once the problem to be solved is decomposed into several modules that can operate in parallel, the programmer has to decide how to map them onto processing elements, which task scheduling policy to use to balance the workload, and how to hide/avoid costly memory accesses or expensive inter-module communications. All these decisions cannot be taken independently from each other because all issues mentioned above are reciprocally connected.

One option to deal with these issues and to reduce the complexity of the parallelization is to introduce *constraints* in such a way that well-known heuristics can be applied to solve the given problem [268, 318]. To this end, one of the most widely acknowledged approaches is the usage of *parallel paradigms* also referred as *parallel patterns*.

Parallel paradigms are schemas of parallel computations that recur in the realization of many algorithms and applications for which parametric implementations are available. Also, such well-known parallel structures have a rigorous semantics with an associated cost model that allows evaluating their profitability. This approach liberates the programmer from the concerns of the *process mapping*, *tasks scheduling*, and *load-balancing*, allowing him/her to concentrate on computational aspects, having only an abstract high-level view of the parallel program, while all the most critical implementation choices are in charge of the programming tools and RTSs.

The programming approach based on parallel patterns is called *structured parallel programming* [122, 123, 94, 242, 318]. This term has been borrowed from sequential programming where in the 60s and 70s programs were often poorly designed. Several computer scientists recognized that programs should organize code more structurally in procedures by using higher-level control structures (e.g., “if-then-else” and “while-do”) aiming to establish structured programming practices [103] expressing a program as the composition of a limited amount of abstract high-level constructs.

The structured parallel programming model provides the parallel application programmer with a set of predefined, ready-to-use parallel abstractions that may be directly instantiated, alone or in composition with, to model the complete parallel behavior of the application. This raises the level of abstraction by ensuring that the application programmer does not need to deal with parallelism exploitation issues and low-level architectural details during application development. Instead, these issues are efficiently managed using state-of-art techniques by the system programmer while designing the development framework and its associated run-time.

More recently, some authors argue that parallel patterns should be used to replace explicit thread programming to improve the maintainability of software [242]. The initial idea was proposed in the late '80s when *algorithmic skeletons* were introduced to simplify parallel programming in the HPC context [93, 267]. As our work directly extends the ideas of algorithmic skeletons, we will discuss them with more details in the following.

2.5.2 Algorithmic Skeletons

Algorithmic skeletons[4], first introduced in the field of High Performance Computing [93, 94], were developed independently of parallel patterns to support programmers with the provisioning of standard programming language constructs that model and implement parametric, and reusable parallel schemes. The skeletons approach inspired the development of several structured parallel programming frameworks and libraries including P^3L [39], ASSIST [317], Muesli [212], SkePU [146], PPL [75] and FastFlow [21]. A survey of the many frameworks developed following the idea of skeletons can be found in González-Vélez and Leyton research work [170].

Meanwhile, the software engineering community extended the classic *design pattern* concepts [162] into the *parallel design pattern* concept [241] inheriting a lot of the algorithmic skeletons ideas and experience. Afterward, the advantages deriving from structured parallel programming approaches have been clearly identified as a viable solution to the development of efficient parallel applications [35, 174, 242]. Formally, a skeleton is a higher-order function that executes one or more user-defined functions following a predefined parallel schema and hiding the details of parallelism exploitation to the user. In a nutshell, algorithmic skeletons may be considered as a practical implementation of parallel patterns [88].

[4] We will refer to "algorithmic skeletons" also as just "skeletons".

2.5.3 A basic set of parallel patterns

Several works have described parallel patterns by providing a formal semantics allowing to compose and nest patterns following rigorous rules [15, 76, 79]. Given the clear functional and parallel semantics of patterns, several rewriting rules have been developed allowing to transform a pattern expression into an equivalent one having the same functional behavior, and an optimized parallel semantics according to some metrics such as for example the number of resources used, the message latency and the overall throughput [13, 173, 14, 63].

In the following, we describe the most common parallel patterns using an informal syntax as presented in our recent work describing the P^{3}ARSEC benchmark suite [129].

Sequential (`seq`). This pattern encapsulates a portion of the business logic code of the application. It can be used as a parameter of other more complex patterns. The implementation requires to wrap the code in a function $f : \alpha \to \beta$ with input and output parameter types α and β, respectively. For each input $x : \alpha$ the pattern $(\texttt{seq}\ f) : \alpha \to \beta$ applies the function f on the input by producing the corresponding output $y : \beta$ such that $y = f(x)$. The pattern can also be applied when the input is a *stream* of elements with the same type. Let $\alpha\ \textsf{stream}$ be a sequence $(x_1, x_2, \ldots,)$ where $x_i : \alpha$ for any i. The pattern $(\texttt{seq}\ f) : \alpha\ \textsf{stream} \to \beta\ \textsf{stream}$ applies the function f to all the items of the input stream, which are computed in their strict sequential order, i.e. x_i before x_j iff $i < j$.

Pipeline (`pipe`). The pattern works on an input stream of type $\alpha\ \textsf{stream}$. It models a composition of functions $f = f_n \circ f_{n-1} \circ \ldots \circ f_1$ where $f_i : \alpha_{i-1} \to \alpha_i$ for $i = 1, 2, \ldots, n$. The pipeline pattern is defined as $(\texttt{pipe}\ \Delta_1, \ldots, \Delta_n) : \alpha_0\ \textsf{stream} \to \alpha_n\ \textsf{stream}$. Each Δ_i is the i-th *stage*, that is a pattern instance having input type $\alpha_{i-1}\ \textsf{stream}$ and output type $\alpha_i\ \textsf{stream}$. For each input item $x : \alpha_0$ the result out of the last pipeline stage is $y : \alpha_n$ such that $y = f_n(f_{n-1}(\ldots f_1(x) \ldots))$. The parallel semantics is such that stages process in parallel distinct items of the input stream, while the same item is processed in sequence by all the stages.

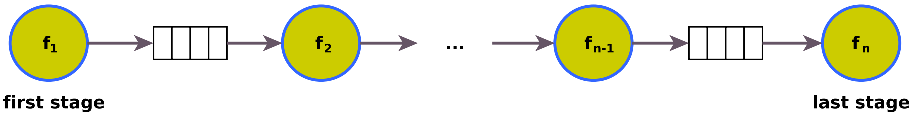


Figure 2-14: Implementation schema of a *pipeline* of n stages.

From an implementation viewpoint, a pipeline of sequential stages is implemented by concurrent activities (e.g., threads), which communicate through FIFO queues carrying messages or reference to messages. Figure 2-14 shows an implementation schema.

Task-Farm (`farm`). It computes the function $f : \alpha \to \beta$ on an input stream α stream where the computations on distinct items are independent. The pattern is defined as (`farm` Δ) : α stream $\to$ β stream where Δ is any pattern having input type α stream and output type β stream. The semantics is such that all the items $x_i : \alpha$ are processed and their output items $y_i : \beta$ where $y_i = f(x_i)$ computed. From the parallel semantics viewpoint, within the farm the pattern Δ is replicated $n \geq 1$ times (n is a non-functional parameter of the pattern called *parallelism degree*) and, in general, the input items may be computed in parallel by the different instances of Δ. In the case of a farm of sequential pattern instances, the run-time system can be implemented by a pool of identical concurrent entities (*worker* threads) that execute the function f on their input items. In some cases, an active entity (usually called *Emitter*), can be designed to assign each input item to a worker, while in other systems the workers directly pop items from a shared data structure. Output items can be collected and their order eventually restored by a dedicated entity (usually called *Collector*) that produces the stream of results (see Figure 2-15).

Master-worker (`master-worker`). This pattern works on a *collection* (α collection) of type α, i.e. a set of data items $\{x_1, x_2, \ldots, x_n\}$ of the same type $x_i : \alpha$ for any i. There is an intrinsic difference between a stream and a collection. While in a collection all the data items are available to be processed at the same time, in a stream the items are not all immediately available, but they become ready to be processed spaced by a certain and possibly unknown time interval. The pattern is defined as

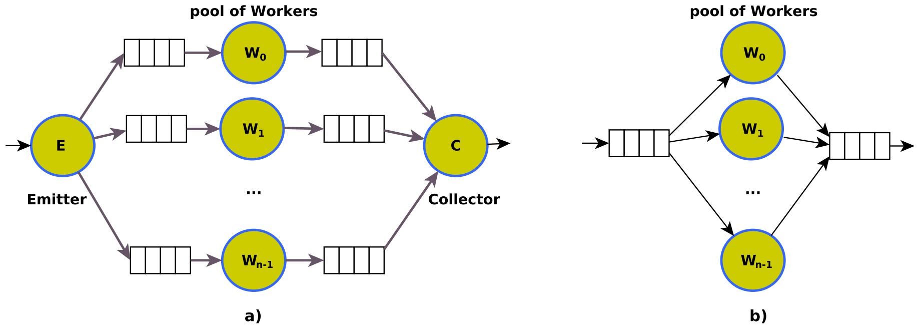


Figure 2-15: Implementations schema of a *farm* of n Workers, with (a) and without (b) the *Emitter* and *Collector* entities.

(`master-worker` Δ, p) : α collection $\rightarrow$ α collection where Δ is any pattern working on an input type α and producing a result of the same type, while p is a boolean predicate. The semantics is that the `master-worker` terminates when the predicate is `false`. Different items can be computed in parallel within the `master-worker`.

A `master-worker` of sequential pattern instances consists of a pool of concurrent workers that perform the computation on the input items delivered by a *master* entity. The master also receives the items back from the workers and, if the predicate p is `true`, reschedules some items.

Map (`map`). The pattern is defined as (`map` f) : α collection $\rightarrow$ β collection and computes a function $f : \alpha \rightarrow \beta$ over all the items of an input collection whose elements have type α. The output produced is a collection of items of type β where each $y_i : \beta$ is $y_i = f(x_i)$. The precondition is that all the items of the input collection are independent and can be computed in parallel.

The run-time of the `map` pattern is similar to the one described for the `farm` pattern. The difference lies in the fact that since we work with a collection, the assignment of items to the worker entities can be performed either statically or dynamically. Depending on the framework, an active entity can be designed to assign input items to the workers according to a given policy.

Map+reduction (`map+reduce`). It is defined as (`map+reduce` f, $\oplus$) : α collection $\rightarrow$ β, where $f : \alpha \rightarrow \beta$ and $\oplus : \beta \times \beta \rightarrow \beta$. The semantics is such that the function f is applied on all the items x_i of the input collection (map phase). Then, the final result of the pattern $y : \beta$ is obtained by composing all the items y_i of the output collection result of the map phase by using the operator $\oplus$, i.e. $y = y_1 \oplus y_2 \oplus \ldots \oplus y_n$.

A typical implementation is the same as the map where the reduction phase can be executed serially, once all the output items have been produced, or in parallel according to a tree topology by exploiting additional properties of the operator $\oplus$ (i.e. if it is associative and commutative).

Composition (`comp`). This pattern models the composition of two pattern instances that work either on single items, on streams or on collections. In case of collections, the composition is (`comp` Δ_1, Δ_2) : α $\mathsf{collection} \to \gamma$ $\mathsf{collection}$ where Δ_1 is any pattern (e.g., `map` or `master-worker`) working on input α $\mathsf{collection}$ and that produces an output β $\mathsf{collection}$, while Δ_2 is a pattern working with input type β $\mathsf{collection}$ and transforming it into a type γ $\mathsf{collection}$. The semantics is that the first pattern is executed, and when its execution has finished (i.e. all the items in the input collection have been computed) the second pattern can be started for processing the collection produced by the first pattern. In case of streams, the composition semantics is applied on an item-by-item basis, i.e. each item in the input stream is processed first by Δ_1 and then by Δ_2 before starting to compute the next item. The RTS of a pattern-based framework must ensure that the two patterns within the `comp` instance are executed serially. In the case of collections, a barrier can be placed between the two patterns.

Iterator (`iterator`). In its basic form this pattern iterates a pattern Δ working on a single input item (`seq` or `comp`) or on a collection of items (`map`, `master-worker`). In case of collections, the pattern is defined as (`iterator` Δ, p) : α $\mathsf{collection} \to \alpha$ $\mathsf{collection}$, where p is a boolean predicate. The inner pattern Δ is iterated until the predicate is `true`. At the implementation level, the run-time executes the pattern for a certain number of times determined statically or at run-time. At the end of each iteration there is an implicit barrier, since the output collection computed at iteration $i - 1$ may be used as input for the iteration i.

2.5.4 Parallel Building Blocks

Different from a pure skeleton based approach where a highly specialized, efficient and monolithic implementation of each skeleton is optimized for a given target architecture, the building block approach, recently proposed [118, 7], pushes the idea of providing the programmer with efficient reusable implementations of basic components that can be assembled following a LEGO-style methodology to build and orchestrate more complex parallel structures. By using RISC-pb^{2}l components, it is possible to model both well-known parallel patterns and general-purpose parallel programming abstractions not usually listed in classical skeleton sets, and also more specialized domain-specific parallel patterns. Borrowing the concept of higher-order functions/combinators distinction proposed for sequential building blocks in Backus' FP programming language [40], the RISC-pb^{2}l approach aims to provide a similar "algebra of programs" that is suitable for reasoning about parallelism to enable refactoring and optimization of parallel programs.

RISC-pb^{2}l is a description language for structured parallel programming built upon three distinct components: *wrappers*, *combinators*, and *functionals*. Wrappers describe how a given function has to be executed. They are used to encapsulate either sequential or parallel code. Combinators are used to establish data-flow connections between building blocks. Functionals are used to represent basic parallel patterns. The complete set of RISC-pb^{2}l building block is shown in Figure 2-16. Each component follows a data-flow semantics.

As an example, both a *Task-farm* and *Map* parallel patterns (with a parallelism degree of n Workers) can be described in RISC-pb^{2}l as follows:

$$\begin{aligned}
\text{Farm(f,n)} &= \triangleleft_{Unicast(Auto)} \bullet [|\ \Delta\ |]_n \bullet \triangleright_{Gather} \\
\text{Map(f,n)} &= \triangleleft_{Scatter} \bullet [|\ \Delta\ |]_n \bullet \triangleright_{Gatherall}
\end{aligned}$$

In both cases, f is the function executed by Δ and the parallel structure is defined by a three-stage pipeline composition of building blocks. The *Auto* policy in the $\triangleright_{Pol}$ building block defines an auto-scheduling policy. It may be substituted by other

Name	*Syntax*	*Informal semantics*
Wrappers		
Seq wrapper	$((f))$	Wraps sequential code into a RISC-pb^{2}l "function" .
Par wrapper	$(\| f \|)$	Wraps any parallel code into a RISC-pb^{2}l "function" (e.g., code offloading to GP-GPU or a parallel OpenMP code).
Functionals		
Parallel	$[\| \Delta \|]_n$	Computes in parallel n identical programs on n input items producing n output items.
MISD	$[\| \Delta_1, \ldots, \Delta_n \|]$	Computes in parallel a set of n different programs on n input items producing n output items.
Pipe	$\Delta_1 \bullet \ldots \bullet \Delta_n$	Uses n (possibly) different programs as stages to process the input data items and to obtain output data items. Program i receives inputs from program $i-1$ and delivers results to $i+1$.
Reduce	$({}_g\triangleright)$	Computes a single output item using an l level ($l \geq 1$) $k-ary$ tree. Each node in the tree computes a (possibly commutative and associative) $k-ary$ function g.
Spread	$({}_f\triangleleft)$	Computes n output items using an l level ($l \geq 1$) $k-ary$ tree. Each node in the tree uses function f to compute k items out of the input data items.
Combinators		
1-to-N	$\triangleleft_{D\text{-}Pol}$	Sends data received on the input channel to one or more of the n output channels, according to policy *D-Pol* with $D\text{-}Pol \in [Unicast(p), Broadcast, Scatter]$ where $p \in [RR, AUTO]$. *RR* applies a round robin policy, *AUTO* directs the input to the output where a request token has been received
N-to-1	$\triangleright_{G\text{-}Pol}$	Collects data from the n input channels and delivers the collected items on the single output channel. Collection is performed according to policy *G-Pol* with $G\text{-}Pol \in [Gather, Gatherall, Reduce]$. *Gatherall* waits for an input from all the input channels and delivers a vector of items, *de facto* implementing a barrier.
Feedback	$\overleftarrow{(\Delta)}_{cond}$	Routes output data y relative to the input data x ($y = \Delta(x)$) back to the input channel or drives them to the output channel according to the result of the evaluation of $Cond(x)$. May be used to route back n outputs to n input channels as well.

Legal compositions grammar

$$\begin{array}{lcl}
\Delta^n & ::= & [\| \Delta \|]_n \quad | \quad [\| \Delta_1, \ldots, \Delta_n \|] \quad | \quad \overleftarrow{(\Delta^n)}_{cond} \quad | \quad \Delta^n \bullet \Delta^n \\
\Delta^{1n} & ::= & \triangleleft_{Pol} \quad | \quad ({}_f\triangleleft) \\
\Delta^{n1} & ::= & \triangleright_{Pol} \quad | \quad ({}_g\triangleright) \\
\Delta & ::= & ((code)) \mid (\| \; code \; \|) \mid \Delta \bullet \Delta \mid \overleftarrow{(\Delta)}_{cond} \mid \Delta^{1n} \bullet \Delta^{n1} \mid \Delta^{1n} \bullet \Delta^{n} \bullet \Delta^{n1}
\end{array}$$

Figure 2-16: RISC-pb^{2}l building blocks and its composition grammar as presented in [7].

policies, such as for example by a simple round-robin policy (*RR*). In the *Map* implementation, a *gather-all* collection policy is used to gather all results. In the *Task-Farm* pattern a simple non-deterministic collection of results is applied.

The RISC-pb^{2}l set has been recently extended to automatically extract the extra-functional properties of an application with the aim of targeting efficient execution on heterogeneous platforms [169]. The new framework, called SKIP (Structural Composition and Interaction Protocol), coordinates the execution of structured parallel applications in heterogeneous multi-cores performing autonomic scheduling of components in heterogeneous devices and application performance related tuning.

2.6 Summary

In this chapter, we have provided an overview of the complex and variegated world of parallel computing. We described the most relevant computing architectures with particular emphasis on the transition from multiprocessors to multi-core and many-core systems. Then we described the most relevant programming models focusing on the structured parallel programming methodology which is at the basis of our work. In our opinion the materials contained in this chapter should help the reader to go through the contributions of the thesis and provide the relevant background to comprehend related work, which is presented in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Overview of Existing Tools

A large body of research considers parallel programming frameworks and tools for parallelism exploitation. Some of them mainly focus on simplifying parallel programming others instead focus more on targeting multiple platforms aiming at increasing portability. Among this latter, some proposals target heterogeneous systems equipped with GPUs and HW accelerators.

In this chapter, we discuss notable projects related to parallel programming proposed mainly by research groups. We do not pretend to be exhaustive nor to discuss all features of the framework or library we present. However, we provide an overview of the most important features that characterize each of them to give an overview of the current state of the art related to parallel programming tools.

We start considering skeleton-based libraries, since the FastFlow framework has been developed following the main principles of the *structured parallel programming model*. Then, we focus on tools offering a high-level approach to parallel programming and annotation based approaches. We also briefly consider existing tools and libraries for programming distributed parallel systems.

Finally, we summarize what emerged from the analysis of existing tools, briefly discussing the most recent research directions.

3.1 Skeletons-based libraries

Skeleton-based programming (often referred to as *structured parallel programming*) has been introduced by Murray Cole in his Ph.D. thesis [93]. Cole defined a skeleton programming environment as follows:

> The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

Since then, the concept of algorithmic skeleton has been used by several research groups to design high-performance structured parallel programming libraries and frameworks with the main aim to replace the traditional low-level programming models with better abstractions and an easier way to express parallelism through a collection of recurrent and efficient parallel exploitation patterns [243, 242, 241]. The main aim of this programming methodology is to provide a good trade-off between *programmability*, *reusability* of code, *portability* and *performance* enhancement to improve programmers' productivity by letting them focus on algorithms and business logic code instead of hardware architectures and low-level details.

Skeletons are provided to the programmer either as language constructs (for example, in the *ASSIST* parallel framework [317]), or as libraries (for example in *SkePU* [146] and *SkelCL* [304] frameworks). They can be nested to build complex parallel applications. Usually, the set of skeletons provided includes both data-parallel, task and stream parallel patterns. The compiling tools of the skeleton language or the RTSs of the skeleton libraries take care of automatically deriving/executing efficient parallel code out of the skeleton application without any direct programmer intervention [268].

Recently, an increasing interest in Domain Specific Languages (DSLs) as a means for tacking *performance portability* led to the proposal of some tools offering *skeletons* for design-space exploration before generating parallel application code for the target platform [121].

In the rest of this section we briefly describe some notable research framework mainly targeting multi/many-core platforms and GPUs. We will discuss skeleton-based frameworks targeting distributed systems in Section 3.4.3.

3.1.1 SkePU

SkePU [146] is an open-source high-level C++ programming framework for heterogeneous parallel systems, with a primary focus on multi-core CPUs and multi-GPU systems. Its main objectives are to enhance both performance portability and to provide a more programmer-friendly interface than low-level APIs such as OpenCL and CUDA. It is implemented as a C++ template library that provides a unified interface for data-parallel computations through *algorithmic skeletons* both on GPUs using OpenCL and CUDA backends and on multi-core CPUs by using a parallel OpenMP backend.

The first version of SkePU (called SkePU-1), developed until 2015 (the latest release was SkePU 2.1), used a macro-based language where C preprocessor macros were used to abstract the target platform. The SkePU-1 user functions, generated from a macro interface, were C++ objects containing member functions for CUDA and CPU targets, and strings of code for the OpenCL target to be dynamically compiled. Deciding which backend to use for a given application, depends upon several different factors such as the problem size the kind of target platforms and the kind of skeleton used. In [125], the SkePU authors proposed an automatic selection algorithm based on offline machine learning algorithm which generates a decision tree with low training overhead that in the end provides the user with an auto-tuning mechanism for backend selection.

SkePU-1 includes different flavors of the *Map* and *Map-Reduce* skeletons (*Map*, *MapArray*, *MapOverlap*, *Reduce*, *MapReduce*), the *Scan* skeleton that is a generalized prefix sum operation with a binary associative operator, and the *Generate* skeleton that allows us to initialize elements of an aggregate data structure based on the element index and a shared initial value.

SkePU-1 includes also aggregate data structures called *smart containers* [124]. Smart containers are concurrent data structures of generic elements (currently available as vectors and matrices) stored in the host main memory, but that can temporarily store subsets of their elements in GPU device memories to optimize memory-to-memory transfers and device memory allocation. Besides, smart containers perform transparent software caching of kernel operands they wrap.

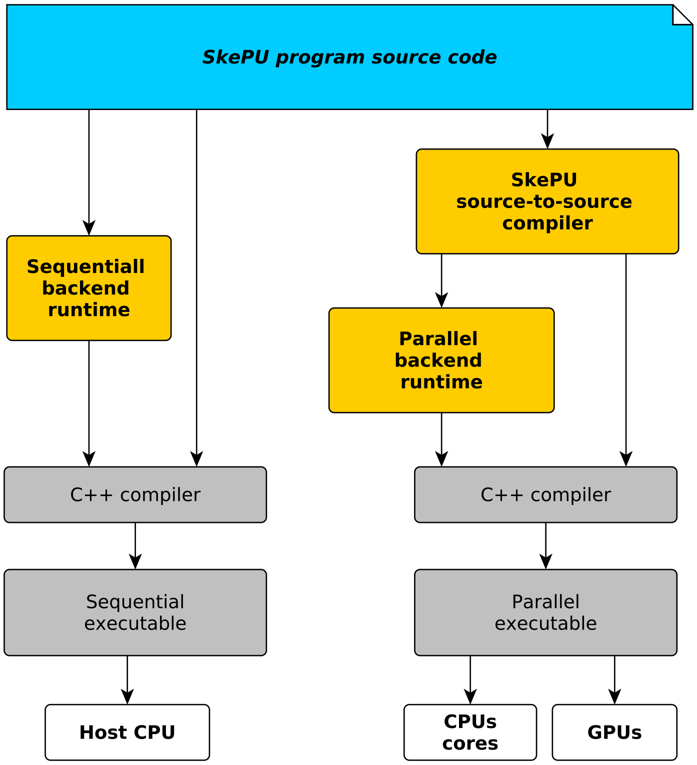


Figure 3-1: The SkePU-2 compiler infrastructure.

SkePU-2 [148] is a redesign of SkePU-1 made available in 2016. It builds on the run-time system of SkePU-1 updated to use C++ variadic template features. It also adds a new user interface based on the modern C++ syntax that leverages lambda expression and a new compilation model with a source-to-source translation. While SkePU-1 uses preprocessor macros to transform user functions for parallel backends, SkePU-2 utilizes a source-to-source precompilation phase based on libraries from the Clang project [277]. The user source code is passed through this tool before the standard compilation phases (see Figure 3-1).

```
 1 // map function working on the single item of the input collection
 2 MapOutput mapFunction(skepu2::Index1D index, parm elem)
 3 { <business-logic code>};
 4 // preparing the input and output data structures
 5 skepu2::Vector<parm> swaptions_sk(swaptions, nSwaptions, false);
 6 skepu2::Vector<MapOutput> output_sk(nSwaptions);
 7 // creating the map object by providing the function to compute
 8 auto map = skepu2::Map<1>(mapFunction);
 9 // setting up the OpenMP backend and the number of threads to use
10 auto spec = skepu2::BackendSpec{skepu2::Backend::Type::OpenMP};
11 spec.setCPUThreads(nThreads);
12 map.setBackend(spec);
13 // map execution invocation
14 map(output_sk, swaptions_sk);
```

Code 1: SkePU-2 implementation of the *swaptions* P^{3}ARSEC benchmark using the OpenMP backend. Code from [129].

The main aim of the redesign of SkePU-2 is to enhance *flexibility* and *type-safety* by removing macros and leveraging C++ features for cleaning up the user interface. SkePU-2 removes the *MapArray* and *Generate* skeletons in favor of a generalized *Map* skeleton and adds the new *Call* skeleton. *Call* is not a skeleton in a strict sense, as it does not enforce a specific structure for computations. It simply invokes its user function. The programmer can provide arbitrary computations as explicit user function backend specializations, which must include at least a sequential CPU backend as a default variant. *Call* extends the traditional skeleton programming model in SkePU with the functionality of user-defined multi-variant components with auto-tunable automated variant selection [148].

An example of SkePU-2 code is shown in Code 1 where a single *Map* pattern is used to parallelize the main kernel of the *swaptions* PARSEC application. The function containing the business logic code operating on each element of the input data is defined in line 2. Input and output data collections are instantiated by using SkePU-2 smart containers (lines 5–6, respectively). Then, the map object is created by providing the map function (line 8) and the OpenMP backend run-time (with its parallelism degree – line 11) is selected for the map pattern at line 10. Finally, the

```
1  int main() {
2   skelcl::init(); // initialize SkelCL
3   // specify calculations using parallel patterns (skeletons):
4   Zip<int(int,int)> mult("int func(int x, int y){ return x*y; }");
5   Reduce<int(int)>  sum("int func(int x, int y){ return x+y; }","0");
6   // create and fill vectors
7   Vector<int> A(1024);
8   Vector<int> B(1024);
9   init(A.begin(), A.end()); init(B.begin(), B.end());
10  // perform calculation in parallel
11  Vector<int> C = sum( mult(A, B) );
12  // access result
13  std::cout << "Dot product: " << C.front() << std::endl;
14 }
```

Code 2: SkelCL program computing the dot product of two vectors. Code from [303].

data parallel computation is executed (line 12).

3.1.2 SkelCL

SkelCL [304] is a high-level programming framework targeting single and multi-GPU systems. It extends OpenCL by introducing algorithmic skeletons operating on container data types. Implemented as a library, it does not require the usage of a precompiler like SkePU, with the downside that user functions are defined as string literals. Parallelism is expressed implicitly, using skeletons, and memory management is performed automatically by the SkelCL run-time built on top of OpenCL. SkelCL aims at simplifying the development of real-world applications for multi-GPU systems while providing performance close to hand-tuned OpenCL implementations. It provides a C++ API that simplifies several repetitive tasks such as explicit data transfer between CPU and GPU, memory allocation and program initialization. SkelCL can also be used in combination with low-level OpenCL code for more advanced programming that requires maximal flexibility.

Thanks to the definition of two container data types (vector and matrix) it automatically moves data between host CPU and GPUs and between multiple GPUs relieving the programmer of all tricky aspects related to the management of multi-GPU platforms. SkelCL run-time abstracts from the physical memory of each GPU in the systems, making available containers' data on each GPU. Code 2 shows a SkelCL program implementing the dot product operation on two vectors using two parallel patterns: *Zip* for computing the *Map* part (line 4) and the *Reduce* to reduce the intermediate results and produce the final result (line 5).

The current version of SkelCL provides four basic skeletons (Map, Reduce, Zip, and Scan) and three more advanced skeletons (Allpairs, MapOverlap, and Stencil). An in-depth discussion of the basic skeletons plus the Allpairs and MapOverlap ones can be found in [303], the Stencil skeleton is discussed in [61]. The AllPairs skeleton is an efficient implementation of specific complex access modes involving multiple matrices.

All computations in SkelCL accept containers as their input and output. To simplify the partitioning of a container on multiple GPUs, SkelCL supports the concept of *distribution* that specifies how a container is distributed among the GPUs. The application developer can set the distribution of containers explicitly or can use the default distribution associated with each skeleton for its input and output containers. The distribution of a container can be changed at run-time (with automatic data exchanges between multiple GPUs and the CPU). Implementing such data transfers in standard OpenCL requires lots of coding which introduces bugs and potential inefficiency. SkelCL hides all this extra low-level coding to the user.

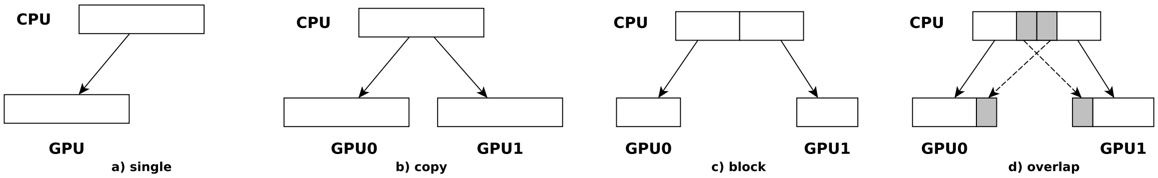


Figure 3-2: Different distribution policies in SkelCL ([61]).

The concept of distribution allows the application developer to abstract from explicitly managing memory ranges which are shared or partitioned across multiple GPUs. Four kinds of distributions are currently available to the application developer in SkelCL: *single*, *copy*, *block*, and *overlap*. For a system with two GPUs, Figure 3-2

```
int main() {
  // initialize Muesli
  msl::initSkeletons(argc, argv);

  auto init = [] (int row, int col)
    { return randomFloat(row, col); };
  // create and initialize a distributed matrix dim x dim
  msl::DMatrix<float> A(dim, dim, 1,
      msl::Muesli::num_total_procs, init,
      Distribution::DIST);
  // create user functions
  auto square = [](float a) {return a*a;};
  auto sum    = [](float a, float b) {return a+b;};
  // apply skeletons
  A.mapInPlace(square);
  float f_norm = A.fold(sum);
  // write result
  printv("||A||_F = %f\n", sqrt(f_norm));
  // terminate Muesli
  msl::terminateSkeletons();
}
```

Code 3: Muesli program computing the Frobenius Norm of a matrix.

illustrates the four kinds of distributions. The overlap distribution is used for the MapOverlap and Stencil skeletons: it stores on both GPUs a common block of data from the border between the GPUs [61].

3.1.3 Muesli

Muesli [212], developed at University of Münster, is a C++ template library that supports both shared-memory as well as distributed memory architectures. It uses MPI for inter-node communications and OpenMP for intra-node parallelism exploitation. Recently, it has also been extended to support multi-GPU systems by using CUDA [147]. It provides data-parallel skeletons such as *Map*, *Fold*, *Scan* (i.e. prefix sum) *zip* and *mapStencil*. *Muesli* also implements distributed data structures such as distributed arrays, matrices, and sparse matrices (in the current version, skeletons operating on sparse matrices cannot be executed on GPUs). Data parallel skeletons are offered to the user as member functions of distributed data structures. Code 3 shows a Muesli implementation of the Frobenius Norm computation for a matrix of size `dim` × `dim` using the *Map* and *Fold* skeletons.

Task parallel skeletons are offered as separate classes. They are used to construct process topologies such as *farm*, *pipeline*, *Divide & Conquer* and *branch-and-bound*. To use a task-parallel skeleton, the user has to instantiate an object of the corresponding class. When nesting distributed data structures into task parallel skeletons, only a subset of processes participate in the task parallel skeleton. In the current version of *Muesli* the programmer must explicitly indicate whether GPUs are to be used for data-parallel skeletons.

3.1.4 Marrow

Marrow [234] is a C++ algorithmic skeleton framework for the orchestration of OpenCL-based computations. It provides a set of data and task-parallel skeletons that can be nested to model complex computations targeting mainly GPU-based systems. Recently, it has been extended to support multiple GPUs [25]. It offers three base skeletons:

- Pipeline: it defines computation with linear data dependencies where the output of a stage is the input of the following one. *Marrow*, implements this pattern by keeping the data buffers needed for inter-stage communications on the GPU memory, thus eliminating host-device memory transfers.

- Loop: This pattern models an iterative loop over the entire skeleton computation until a user-defined condition is evaluated to true. It is particularly useful for numerical methods and scientific simulations which typically are iterative computations. The condition itself may be controlled by external or internal events to the loop body. A particular instance of the *Loop* pattern is the *For* skeleton.

- Map: It applies a computation to independent partitions of the input dataset. This pattern fits particularly well with the semantics of the GPU's execution model. *MapReduce* extends *Map* by allowing the definition of a reduction stage, to be subsequently performed at the host side. Optionally, this reduction step can be partially performed on the GPU. For that purpose, the framework deploys a pipeline, whose second stage is an explicitly supplied OpenCL reduction kernel.

Recently Marrow has been extended to process streams of hybrid CPU/Xeon Phi nodes [155]. Marrow's skeletons have been used to transparently manage communications between the host and the Xeon Phi accelerators.

3.1.5 GrPPI

GrPPI (Generic and reusable Parallel Pattern Interface) is a programming interface for modern C++ applications developed at the University Carlos III of Madrid [135]. It is an open source library that accommodates a layer between application developers and existing parallel programming frameworks targeting multi-core systems.

According to GrPPI authors, the lack of a common terminology to denote different kind of patterns and the lack of a common, recognized and assessed API to use these patterns has prevented the wide diffusion of the pattern-based parallel programming as well as a general acknowledgment of the related advantages. By making use of advanced C++ features such as meta-programming concepts, and generic programming techniques, GrPPI provides a fully C++ compliant API to well-known parallel patterns on top of different programming models having as a result a unified standard interface. GrPPI currently suppotrs as run-times ISO C++ Threads, OpenMP, Intel TBB, and recently FastFlow. Figure 3-3 shows the general view of the GrPPI library.

GrPPI provides four main components to provide a unified interface for the supported frameworks: *i)* type traits, *ii)* pattern classes, *iii)* pattern interfaces and *iv)* execution policies. Type traits are used for function overloading and to allow composition of different patterns. Moreover, by using the `enable_if` type trait from the C++ standard library, functions that are not used are removed at compile time so that only functions meeting the conditions become available to the compiler, thus

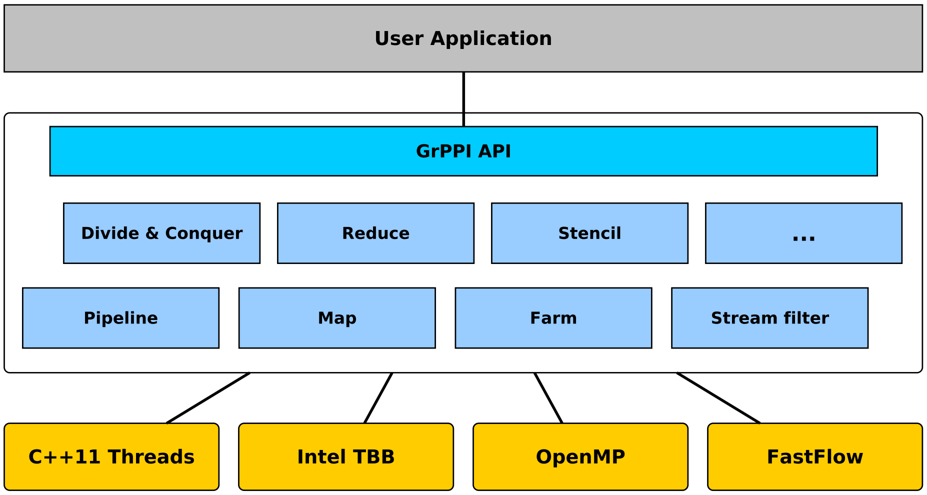


Figure 3-3: The GrPPI software layers.

```
void exec_pipeline(grppi::polymorphic_execution& e,
                   int N) {
 grppi::pipeline(e,
   [N]() -> optional<double> {
     static int x = 0;
     if (x<N) return x++; // produce an element
     else return {};      // end-of-stream
   },
   [](double x) { return F(x); }, // apply F
   [](double x) { return G(X); }, // apply G
   [](double x) { cout << x << endl; } // print result
 );
}
```

Code 4: GrPPI pipeline computing $F(G(x))$ for a stream of N elements.

allowing to provide the same interface for different implementations.

GrPPI provides a set of independent classes that represent each of the supported patterns. These objects store references to other functions and to non-functional information (e.g., concurrency degree) related to the pattern configuration. Thus, they can be used inside another pattern in order to express complex constructions that can not be represented by leveraging a single pattern.

For each supported parallel pattern, GrPPI offers two different alternatives, one for pattern execution and one for composition with other patterns. Both alternatives receive the user functions that will be executed accordingly to the pattern and configuration parameters as function arguments. Code 4 shows a simple GrPPI pipeline computing $F(G(x))$ over a stream of N elements generated by the first stage.

Finally, a key point of GrPPI is the ability to easily switch between different programming framework implementations of the same pattern. This is achieved by providing a set class that encapsulates the actual pattern implementations of a given framework. This way, it is straightforward to use different RTS frameworks. The current GrPPI version provides support for sequential, C++ threads, OpenMP, Intel TBB and FastFlow frameworks.

GrPPI targets the following stream parallel processing patterns: *Pipeline*, *Farm*, *Filter*, *Stream-Reduce*, and *Stream-Iteration*. Recently, GrPPI has been extended with several advanced patters targeting Data Stream Processing (DSP) [136].

3.1.6 Parallel Pattern Library (PPL)

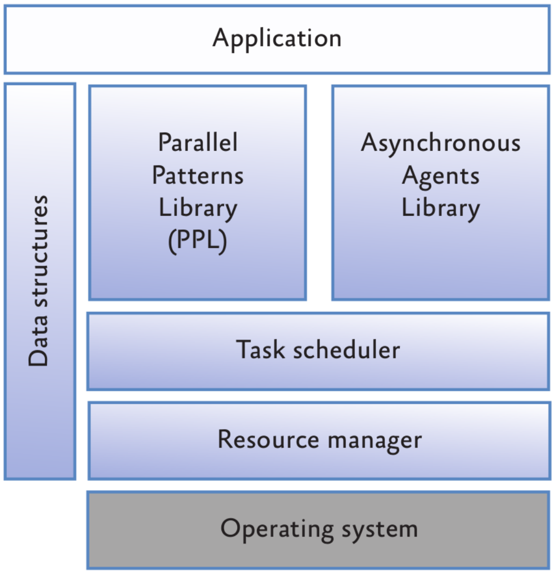


Figure 3-4: Libraries and run-time components of the Parallel Pattern Library (PPL). Figure from “Parallel Programming with Microsoft Visual C++” [75].

The parallel programming support in the Microsoft Visual C++ development system is commonly referred to as the *Parallel Patterns Library* (PPL) [75]. PPL together with the Asynchronous Agents Library (AAL) aim at simplify the process of adding parallelism and concurrency to applications written in C++. It raises the level of abstraction between the application code and the underlying threading mechanisms by providing generic, type-safe algorithms and containers that operate in parallel.

The library provides an imperative programming model that promotes parallel patterns to improve scalability and simplify the development of concurrent applications. To abstract low-level threading details and architectural details, PPL and AAL use the *Concurrency Runtime* which is responsible of task scheduling and resource management (see Figure 3-4). The task scheduler determines on which cores and when to run an application's tasks. It uses cooperative scheduling to provide load balancing across different cores. The resource manager assigns computing resource to the application trying to prevents (or minimize) contention in the use of the cores. It also helps to ensure the best use of cache hierarchy.

PPL provides task parallelism, parallel patterns and data containers that allow safe concurrent access to their elements. PPL currently supports six patterns: *Parallel Loop*, *Parallel Task*, *Parallel Aggregation*, *Futures*, *Dynamic Task Parallelism* and *Pipeline*.

3.2 Other High-Level Approaches

In recent years, in addition to skeleton/pattern-based approaches, a number of frameworks, targeting both shared-memory and distributed systems, have been proposed. For instance, Intel Threading Building Blocks (TBB) [282] for general-purpose parallel computations and StreamIt [310] specifically targeting streaming computations. We consider such approaches as high-level models because their primary aim is to hide some low-level parallel programming details behind suitable APIs or abstractions. In the following we provide a brief overview of some of these high-level approaches to parallelism.

3.2.1 Delite

In the context of parallel programming targeting heterogeneous architectures, is emerging an increasing interest in Domain Specific Languages (DSLs). The main reason is that DSLs can incorporate high-level parallel constructs and domain-specific features together in the same language which facilitates performance portability on a variety of hardware. Compiler analysis in DSLs is typically less conservative than in general-purpose compilers. Some restrictions usually enforced in general-purpose compilers and programming models can be removed allowing to introduce more advanced optimizations [82].

Despite the potential benefits of performance-oriented DSLs, the cost of developing a DSL is high and requires expertise in multiple areas of computer science. Delite is a framework for helping programmers in building compilers for high-performance DSLs targeting heterogeneous architectures (multi-cores, GPUs, clusters and FPGAs) [305]. It is aiming to simplify DSLs construction by providing common reusable components such as parallel patterns, optimizations techniques and code generation features that can be used as basic building blocks by the DSL designer to build high-performance DSLs with low effort in different application domains.

The main components of the Delite framework are shown in Figure 3-5. Delite has been used to develop a suite of DSLs for data analysis: query processing (OptiQL), machine learning (OptiML), graph processing (OptiGraph) and mesh-based PDE solvers (OptiMesh). Delite-generated DSLs (e.g., OptiML), are embedded in Scala [262], a general-purpose functional programming language with a rich type system. Delite compiles DSLs to a platform-neutral Intermediate Representation (IR) by using a Lightweight Modular Staging, an extensible run-time compilation framework that serves as a common basis for all our DSLs [284]. From the IR and by leveraging on knowledge conveyed in the DSL, the run-time code is generated for the target platform. Interestingly, Delite-generated C code has been also used as input to C-based High-Level Synthesis tools, such as Xilinx Vivado [326], to generate better register-transfer level (RTL) implementations [276].

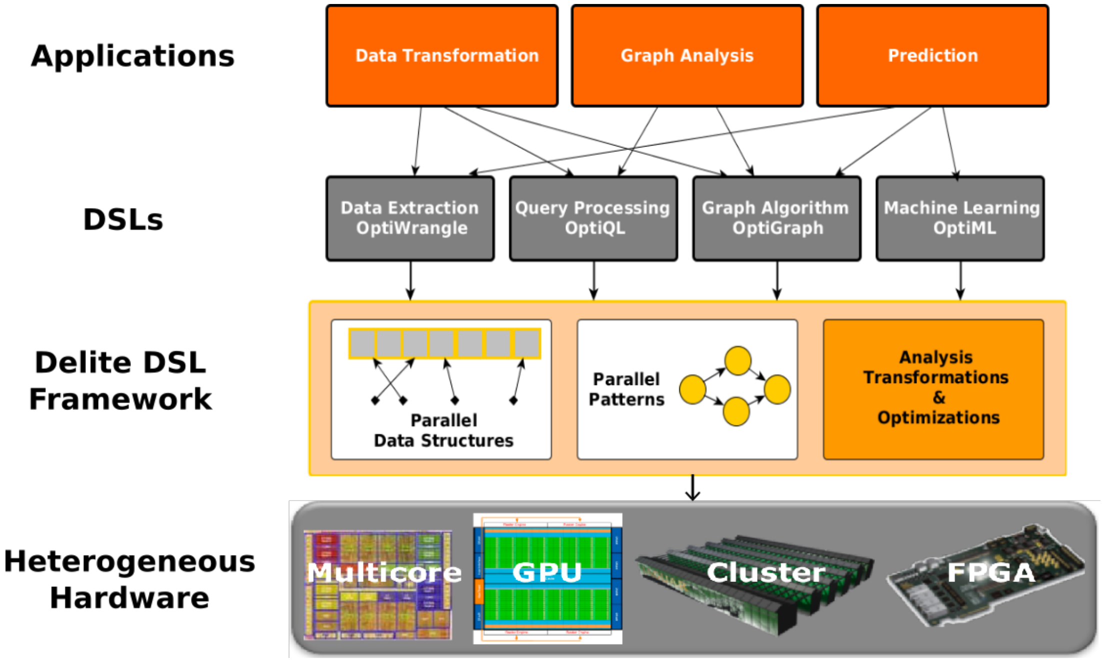


Figure 3-5: The Delite DSL Framework.

The key reason why Delite is able to generate efficient code for the target machine is its emphasis on parallel patterns. While DSLs are exposed to the users mainly to provide more familiar domain-specific abstractions, parallel patterns are used as building blocks in its RTS [305]. Delite currently supports the following parallel patterns: *Sequential*, *Map*, *FlatMap*, *Reduce*, *ZipWith*, *Foreach*, *ForeachReduce*, *Filter*, *GroupBy*, *Sort*. New patterns can be easily added in Delite. In fact, most of the existing Delite patterns extend a common loop-based building block called *MultiLoop* that is a highly generic data-parallel loop.

The Delite platform and associated DSLs are available as open source[1].

3.2.2 Threading Building Blocks (TBB)

Threading Building Blocks (TBB) [282] is a parallel library developed by Intel. The TBB programming model supports task-based parallel programming. It provides the programmer with a collection of thread-safe and efficient concurrent data structures (e.g., concurrent_queue, concurrent_hash_map), a small set of parallel patterns includ-

[1] `http://stanford-ppl.github.com/Delite`

```
void runPipeline(int ntokens, FILE* infile, FILE* outfile) {
  tbb::parallel_pipeline(ntokens,
        tbb::make_filter<void,MyTuple*>(
            tbb::filter::serial_in_order, F1(infile))
    &
        tbb::make_filter<MyTuple*,MyTuple*>(
            tbb::filter::parallel, F2() )
    &
        tbb::make_filter<MyTuple*,void>(
            tbb::filter::serial_in_order, F3(outfile)));
}
```

Code 5: A three-stage pipeline built using TBB.

ing *Map*, *Reduce*, *parallel_for* and *task-graph* and *parallel pipeline*. TBB can be used with any compiler supporting ISO C++ and the most recent versions deeply uses features of modern C++ like *lambda expressions*. TBB exploits the concept of task as unit of parallelism without exposing threads to the programmer aiming at using the available core resources more efficiently. As with Cilk Plus (see Section 3.2.3), TBB implements a thread pool of workers and balancing their workload via *work-stealing* algorithm [52]. The main advantage of this model of execution is that it enables nested parallelism while avoiding some of the issues related to using more threads than available resources (*over-subscription*).

TBB provides also a set of useful classes for example those implementing a lock-free scalable memory allocator, and mutual exclusion mechanisms. Since global locks serialize parts of programs, the TBB implementation, generally avoids locks in its implementation and there is no global tasks' queue.

In Code 5 is shown the top-level code for building and running a TBB pipeline composed of three stages: a sequential stage running function `F1`, a parallel stage running the function `F2` and a final third stage running the function `F3`. The filters are concatenated with the C++ operator '&'. Each filter is constructed by the function `make_filter<inType,outType>(mode, functor)`. The parameter `ntoken` to method `parallel_pipeline` controls the maximum number of in flight data-items (or tokens) processed by the pipeline. In a serial in-order filter, each token is processed serially. In a parallel filter, multiple tokens can by processed in parallel by the filter. Once this limit is reached, the pipeline never creates a new token at the first filter until another token is consumed by the last filter.

3.2.3 Cilk and Cilk Plus

Cilk and Cilk Plus (or Cilk++) are general purpose programming languages designed for multi-threaded parallelism based on C and C++ languages, respectively. Cilk Plus has been released both as open-source as well as a commercial product by Intel.

The *Cilk* research project was initially proposed in the mid 90s [51]. It was released as a C language extension offering keywords such as `cilk_spawn`, `cilk_sync` to spawn the execution of a function (i.e. a task) in a concurrent thread and synchronize with its termination. In Cilk several functions can be spawned at the same time and then their reconciliation can happen at specific synchronization points. This behavior models a `fork-join` pattern of execution where the main control flow forks into several parallel flows of execution that can rejoin in a common synchronization point. The parallel flows can either execute replicas of the same flow or a set of different flows. The nesting of multiple fork-join patterns in a structured fashion generates a hierarchical task-graph.

Cilk Plus evolved from Cilk and currently provides support for both C and C++. Cilk Plus has introduced the explicit specification of vector parallelism through a set of array notations and elemental functions. Moreover, to express loop parallelism, a new keyword `cilk_for` has been added to transform a sequential loop into a parallel loop. The implementation of `cilk_for` loops uses a recursive fork-join approach. A reducer *hyperobject* is a Cilk Plus linguistic construct that allows implementing reductions on a shared variable or data structure. There are several reducers built into Cilk Plus for common reduction operations. It is possible to define a custom reducer for any data type and operation that form a mathematical *monoid* [242].

One of the most interesting features of Cilk was its `work-stealing` task-based scheduler used to balance the workload among a pool of Workers [52]. The fundamental idea of this algorithm is to remove the bottleneck of a single global task queue by using a local task queue for each Worker. Workers obtain tasks only from their queue until it becomes empty. Then, the generic Worker acts as a thief, i.e. selects one of the other workers as a victim (usually randomly) and then tries to steal one or more work items from its queue. This strategy minimizes communication among threads and maximizes local reuse of data. A downside of the work-stealing algorithm is its termination phase or, more generally, the efficient detection of idle states [320]. This is because Workers have only local knowledge. When the local queue runs out of tasks, the Worker is not aware if there are jobs to steal in the other queues. Moreover, if there are very few jobs in the system, it is hard to decide if it is more convenient to keep trying to steal jobs or instead if it is more convenient to terminate.

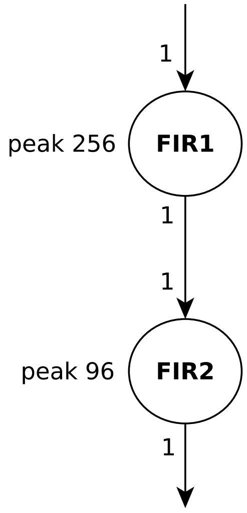
3.2.4 StreamIt

StreamIt is a programming language for high-performance streaming computations [310]. Programs in *StreamIt* are represented as graphs whose nodes (called filters or actors) encapsulate computation, and edges represent inter-filter communications implemented by using FIFO message channels. The programming model of *StreamIt* computation is the Synchronous Data-Flow model (SDF) [219] that is a subset of the pure Data-Flow model (see Section 2.4.4) in which the number of data elements (tokens) produced and consumed by each filter is known at compile-time.

Each filter consists of a work function that repeatedly executes when sufficient data is available on its input FIFO queue. The work function reads data from its input queue, and writes data to its output queue. The work function can also inspect input without removing items from the queue thus allowing to avoid using internal filter state. *StreamIt* provides three basic constructs for composing filters into larger streaming graphs: *pipeline*, *splitjoin*, and *feedback loop*. A *pipeline* connects streams sequentially, a *splitjoin* models a task-farm pattern where streams diverge from a common splitter and merge into a common joiner. A *feedback loop* allows us to create cyclic data-flow graphs.

Filters in *StreamIt* can only access their locally declared variables, therefore data-exchange between filters is accomplished using explicit data transfer. *StreamIt*'s filters

```
void->void pipeline IIR {
  ...
  add FIR(256); add FIR(96);
  ...
}
int->int filter FIR(int n) {
  int w[n];
  ...
  work pop 1 push 1 peek n {
    int i;
    int sum=0;
    for(i=0;i<n;i++)
      sum += peek(n) + w[i];
    pop();
    push(sum);
  }
}
```



Code 6: Example of a 2-filter pipeline in StreamIt. Code from [213].

may be either stateful or stateless. A stateful filter that modifies local state during the work function cannot be parallelized as the next invocation depends on the previous invocation.

A special method called *work* is used to specify the work function that is executed when the filter is invoked in steady state. The stream rates (i.e. the number of items pushed and popped on every invocation) of the work functions are specified statically in the program. The rate-matching guarantees that, during the steady state phase, the number of data elements that is produced by a filter is equal to the number of data elements its successors will consume.

Code 6 shows a simple example of a *StreamIt* program implementing a pipeline of two filters. *StreamIt* provides the *peek* primitive to the programmer, which can be used to non-destructively read values off the input channel. Peeking is an example of a state representation (i.e. sliding window buffer) present in the language. A program using the *peek* instruction can always be rewritten with just pushes and pops plus some local state that holds a subset of values seen so far by filters.

StreamIt also provides a compiler infrastructure to implement the programming language. The compiler can generate code for distinct hardware devices and to optimize code generation through the implementation of data-flow analysis techniques [333, 171].

3.2.5 PACXX

PACXX (Programming Accelerators with C++) is a unified programming model for systems with GPU accelerators developed at the University of Müenster [183]. In PACXX, both host and device programs are written in C++14 standard using data structure implementation from the Standard Template Library (STL). It leverages on all modern features such as variadic templates, lambda expressions and the newly proposed parallel extensions of the STL [97].

PACXX shares many fundamental choices with SkePU 2, for example the fact that the implementation is based on Clang and LLVM frameworks. The authors implement a custom compiler and a run-time system that together perform memory management and synchronization automatically and transparently for the programmer. PACXX includes an easy-to-use and type-safe API for multi-stage programming, memory management is implicitly managed by the compiler and run-time system, which allows the compiler to apply aggressive optimizations [184].

Recently *PACXXv2* has been released with a novel CPU backend which provides portable and predictable performance on various state-of-the-art CPU architectures [185]. It integrates the Region Vectorizer (RV)[2] which is a vectorizer capable of vectorizing any code region in LLVMs intermediate representation.

3.2.6 PiCo

In the context of applications for data analytics, with particular focus on Big Data processing, *PiCo* (*Pi*peline *Co*mposition) has recently been proposed as a data-flow model aiming at expressing Big Data processing applications in terms of graphs of functional-style operators [250, 251]. The main entity of this programming model

[2] RV: A Unified Region Vectorizer for LLVM – `https://github.com/cdl-saarland/rv`

is the *Pipeline*, basically a Direct Acyclic Graph (DAG)-composition of processing elements. This model is intended to give the user a unique interface for both stream and batch processing, hiding data management and focusing only on operations, which are represented by Pipeline stages. PiCo is built on top of the FastFlow library, and currently targets shared-memory platforms.

At lower-level, PiCo implements an application as a composition of FastFlow *farm*s and *pipeline*s. In particular, the *farm* pattern is exploited to express the parallelism both between operators and within an operator. Data collections to be processed, either bounded data sets or unbounded streams, are split into micro-batches, that are streamed to the application as atomic data elements. Of particular interest is the proposal of removing points of centralization (i.e. *farm*'s Emitter and Collector) by refactoring the application graph and introducing a shuffling data operation [250].

3.2.7 C++ Actor Framework (CAF)

The C++ Actor Framework (*CAF*) [87] enables the development of concurrent programs based on the Actor model leveraging on modern C++ language. Different from other well-known implementations of the Actor model, such as Erlang [32] and Akka [199], which use virtual machine abstractions, *CAF* is entirely implemented in C++, and thus applications implemented in *CAF* are compiled directly into native machine code. This allows use of the high-level programming model offered by actors without sacrificing performance introduced by virtualization layers.

CAF applications are built decomposing the computation in small independent work items that are spawned as actors and executed cooperatively by the *CAF* run-time. Actors are modeled as lightweight state machines that are mapped onto a pre-dimensioned set of run-time threads called *Workers*. Instead of assigning dedicated threads to actors, the *CAF* run-time includes a scheduler that dynamically allocates ready actors to Workers. Whenever a waiting actor receives a message, it changes its internal state to *ready* and the scheduler assigns the actor to one of the Worker thread for its execution. As a result, the creation and destruction of actors is a lightweight operation.

Actors that use blocking system calls (e.g., I/O functions) can suspend run-time threads creating either imbalance in the threads workload or starvation. The *CAF* programmer can explicitly *detach* actors, so that the actor lives in a dedicated thread of execution. A particular kind of detached actor is the *blocking actor*. Detached actors are not as lightweight as *event-based* actors.

In *CAF*, actors are created using the `spawn` function. It creates actors either from functions/lambdas or from classes and returns a network-transparent actor handle. Communication happens via explicit message passing using the `send` command. Messages are buffered in the mailbox of the receiver actor in arrival order before they are processed. The response to an input message can be implemented by defining *behaviors* (usually through C++ lambdas). Different behaviors are identified by handler function signature, for example using *atoms*, i.e. non-numerical constants with unambiguous type.

3.2.8 Multi-core Task management API (MTAPI) and EMB^2

The Multicore Association (MCA) has been set-up by a group of leading-edge companies for addressing the programming challenges of heterogeneous embedded multi-core platforms (`https://www.multicore-association.org/`). The main objective of the association is the definition of a set of open specifications and APIs to facilitate multi-core software development and to improve development of portable code across different kind of multi-cores. MCA offers several industry-standard APIs such as the Multicore Resource Management API (MRAPI) for data sharing among different types of cores, the Multicore Communication API (MCAPI) for inter-core communication, and for task management the Multicore Task Management API (MTAPI).

Of particular interest is the MTAPI effort. MTAPI decomposes computations into multiple tasks, then schedules them among the available processing units, and combines the results after specific synchronization. The main components of MTAPI are:

- *Node*: An MTAPI node is an independent unit of execution. A node can be a process, a thread, a thread pool, a general purpose processor or an HW accelerator (e.g., DSP).

- *Job* and *Action*: A job is an abstraction representing the work and is implemented by one or more actions depending on, for example, by the target node which will execute the job (e.g., CPU or GPU). The MTAPI system binds tasks to the most suitable actions during run-time.

- *Task*: An MTAPI task is an instance of a job together with its data environment. Tasks are very light-weight entities that can be created, scheduled so that multiple tasks can be executed in parallel. Depending on the action bindings, a task can be offloaded to a node.

- *Queue*: A queue is defined by the MTAPI specification to guarantee sequential execution of tasks.

- *Group*: MTAPI groups are defined for synchronization purposes. A group is similar to the concept of a synchronization barrier. Tasks attached to the same group must be completed before starting the next step.

MTAPI provides a dynamic binding policy between tasks and actions. This is mainly thought to facilitate jobs to be scheduled on more than one HW accelerator. In this model, the task scheduler plays a central role for the efficient execution of tasks on the heterogeneous multi-core platform.

Siemens, as one of the members of the MCA consortium, created an industry-grade MTAPI implementation as part of a larger open source project called *Embedded Multicore Building Blocks* (*EMB*^{2}) [290]. Besides the task scheduler which provides the Multicore Task management API (MTAPI), EMB2 provides also a set of parallel building blocks and concurrent data structures specifically designed for embedded systems with particular attention to their typical requirements such as predictable memory consumption, and real-time capability (see Figure 3-6).

The EMB2 library supports task priorities and affinities, and the scheduling strategy can be optimized for minimal latency and fairness. It also provides high-level parallel patterns for implementing stream processing applications (such as *pipeline*

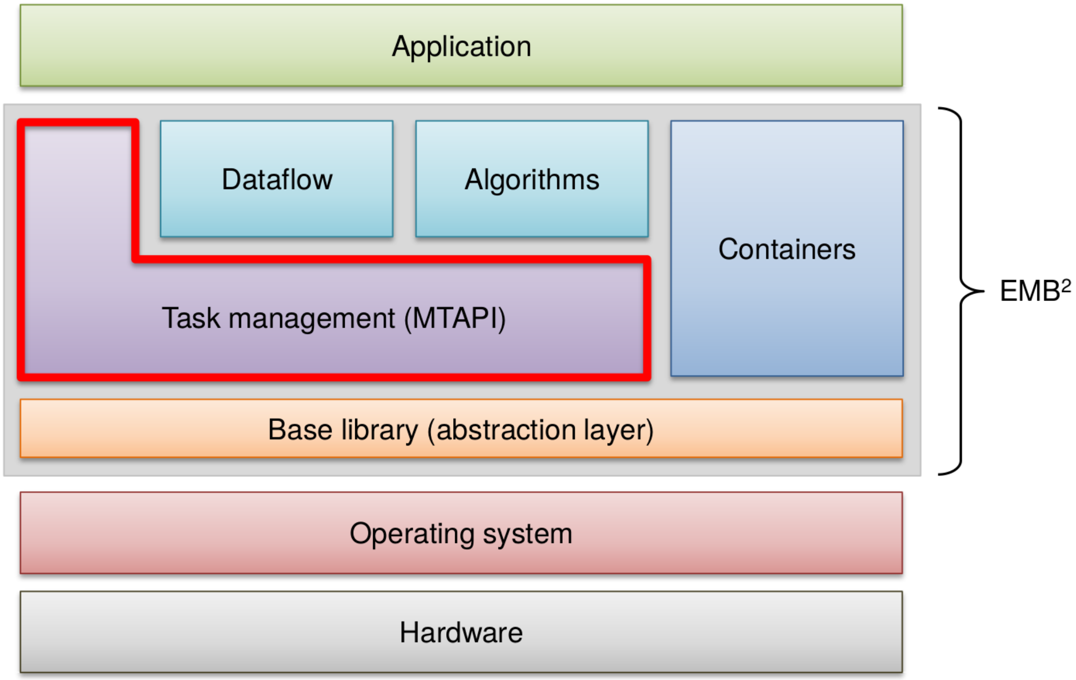


Figure 3-6: The EMB^2 framework using the Task management API (MTAPI). Figure from [290].

and *Task-Farm*), parallel algorithms for parallelizing loops with reduction variables and a small set of concurrent data structures implemented in a non-blocking fashion to minimize locking overhead and guarantee program progress [252].

3.3 Annotation-based approaches

3.3.1 OpenMP

OpenMP (Open Multi Processing) [102, 84] is considered the `de facto` standard API for programming CMPs. It is an open specification (currently is at version 4.5) that defines language extensions for expressing task and data parallelism based on compiler directives (within the C/C++ languages, directives are referred to as *pragmas*). The OpenMP API is supported by major C, C++ and Fortran compilers. Compilers that do not support specific pragmas, merely ignore them so that an OpenMP program can be compiled and executed on every system with a standard sequential compiler. In addition to compiler directives, OpenMP also includes a small set of run-time library routines and environmental variables that are used to modify some non-functional features, as the degree of parallelism used in different portions of the program or the placement of RTS threads on machine cores (thread-to-core affinity).

OpenMP promotes the `fork-join` execution model. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. The main program (i.e. `master-thread`) is run sequentially while specific regions of code, called `parallel regions`, are "accelerated" by spawning a team of parallel threads which execute an instance of the parallel region. At the end of the parallel region, all threads synchronize by joining the master-thread. The compiler directives are used to describe parallel sections and declare which variables are shared or private.

OpenMP offers a higher level parallel programming model compared to Pthreads. OpenMP allows declaring which block of code should be executed in parallel, leaving to the compiler and the RTS the responsibility to deal with the details of the thread execution and synchronization. Furthermore, it enables incremental parallelization of existing sequential code by simply adding compiler directives, allowing a smooth transition from sequential code to parallel code. This was one of the keys of success of OpenMP. However, using a sequential application to generate a parallel one is usually limiting the potential parallelism that can be exploited.

One of the primary sources of parallelism in OpenMP are program loops. The focus on loops is based on the observation that loops often iterate over large data sets performing operations on every data item without any dependencies between iterations. In OpenMP, the programmer annotates such loops with compiler directives, and a compiler supporting the OpenMP standard automatically generates code for executing the loop in parallel. OpenMP implementations do not check that loop iterations are independent or that race conditions do not exist. As in other frameworks (e.g., Cilk and TBB) implementing correct parallelizations is the responsibility of the programmer.

In addition to the `parallel_for` construct, which can be used to express loop parallelism, OpenMP has been enriched with pragmas targeting task parallelism, allowing the user to identify which block of code should be considered as a separate task, leaving to the RTS the burden of efficient scheduling and execution of tasks.

Starting with OpenMP 4.0 it is possible to offload parallel regions to HW accelerators such as GPGPUs. Here the programmer has to additionally specify the data regions which should be copied to and from the GPU before and after the computation. The new version introduced the `target` directive, which denotes a region of a code that will be directly mapped onto the device for execution. Though not as mature as the CPU version, targeting GPUs with OpenMP seems a promising approach to heterogeneous parallelism [235].

3.3.2 OmpSs

OmpSs [144] is a programming model developed at the Barcelona Supercomputer Center and based on OpenMP and StarSs models. It is a framework focusing on the task-based programming model for developing parallel applications on heterogeneous multi-cores. The StarSs model (Star Superscalar) [274] has been used for general-purpose multi-core platforms in the SMP Superscalar framework (SMPSs) [271] and also for system equipped with multiple GPUs (GPU Superscalar – GPUSs [37]). More recently the StarSs model combined with the OpenMP pragma-based approach has been integrated into a single infrastructure targeting heterogeneous multi-cores with the name OmpSs.

StarSs like OpenMP enables programmers to express parallelism by adding pragmas to their sequential code. These pragmas identify regions of code that can be executed as tasks once their input data are ready. The programmer, does not have to explicitly express data dependencies and synchronizations between task. Dependencies are deduced from data accesses by means of task annotation clauses: *input*, *output* and *inout*. Using these directionality clauses, the programmer specifies which data each task accesses and how the data is accessed (read, write, or both). OmpSs then uses this information to build a task dependency graph at run-time. Besides performing a task-based parallelization, the run-time system moves the data as needed between the different GPUs minimizing the impact of communication by using afnity scheduling and by overlapping communication with the computation of tasks.

```
#pragma omp task inout ([TS][TS]A)
void spotrf(float *A);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm(float *T, float *B);
#pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
void sgemm(float *A, float *B, float *C);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk(float *A, float *C);

void cholesky(int NT, float *A[NT][NT]) {
  for(int k=0;k<NT;k++){
    spotrf(A[k*NT+k]);
    for(int i=k+1;i<NT;i++)
      strsm(A[k*NT+k],A[k*NT+i);
    for(int i=k+1;i<NT;i++) {
      for(int j=k+1;j<i;j++)
        sgemm(A[k*NT+i],A[k*NT+j],A[j*NT+i]);
      ssyrk(A[k*NT+i],A[i*NT+i);
    }
  }
}
```

Code 7: Cholesky algorithm implementation using OmpSs annotations. Code from [167].

OmpSs uses a thread-pool execution model instead of the traditional OpenMP fork-join model. The master thread starts the execution and all other threads cooperate executing the work it creates. Nesting of constructs allows other threads to generate tasks as well. OmpSs assumes that multiple address spaces may exist. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and shared data which has been marked explicitly with data specification directives (*copy_in*, *copy_out*, *copy_inout* and *copy_deps*. This assumption is true even for SMP machines as the implementation may reallocate shared data to improve memory accesses on NUMA architectures.

OmpSs enables incremental parallelization, where the source code is restructured and optimized step-by-step, while the architecture specific details are separated by the implementation. It is designed to be portable as the same pragmas can be potentially used by any host language and target any architecture that has an implemented backend.

Code 7 shows an example of OmpSs application. The example implements a Cholesky factorization algorithm. The kernels of the factorization have been annotated with OmpSs compiler directives. The directionality clauses indicate whether the given parameter is read, write or read and write in the scope of the task.

3.3.3 StarPU

StarPU [36] is a C-based RTS for scheduling a graph of tasks onto a heterogeneous multi-core machine. It is primarily meant to be used as a backend of a parallel programming framework. It promotes task-based parallel programming following two basic principles: i) tasks may have several implementations on the basis of the various heterogeneous processing units available in the target machine, and ii) data-transfers and synchronizations of different tasks are handled transparently by the RTS.

StarPU uses the concept of *codelet*, a C structure containing different implementations of the same functionality for different target devices (e.g., CPU and GPU). A StarPU task is an instance of a *codelet* applied to some data. The programmer asynchronously submits all tasks by registering all the input and output operands needed to compute the tasks. Tasks that depend on other tasks can either be specified by the programmer using explicit *tags* or can be derived automatically by StarPU considering data dependency rules (e.g., read-after-write).

StarPU schedules tasks at run-time by moving data automatically and transparently to the programmer. There are several built-in scheduling (e.g., work-stealing) and selection policies also based on task priority. Moreover, to avoid unnecessary transfers, StarPU keeps data where it was last needed, even if it was modified there, and it allows multiple copies of the same data to reside at the same time on several processing units as long as it is not modified. StarPU RTS implements a Software DSM with relaxed consistency model. The programmer has to register the different pieces of data by giving their addresses and sizes in the main memory. Data can thus be dynamically as well as statically allocated.

3.3.4 OpenACC

OpenACC is an industry standard proposed for heterogeneous computing (http://wwww.openacc.org). It is a directive based language abstracting parallel programming by using pragma annotations, relieving programmers from specifying how codes should be mapped onto the target platform. It was introduced to manage parallelism on accelerators, such as GPUs, although the same code can be compiled and run also on standard CPU multi-cores.

OpenACC follows the OpenMP approach for introducing parallelism via sequential code annotation of compiler directives with the aim to target multiple hardware accelerators. These directives identify which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself.

The execution model targeted by OpenACC implementations is the host-directed execution targeting multiple attached accelerator devices (e.g., GPUs). User's applications execute on the host while compute intensive regions of code are offloaded to the accelerator device under control of the host. The device executes either parallel regions, which typically contain loops that are executed in parallel on the accelerator, or serial regions, which are blocks of sequential code that execute more efficiently on the accelerators. Even in accelerator-targeted regions, the host may orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. In most cases, the host can queue a sequence of operations to be executed on the device, one after the other [263].

Though the best performance is typically achieved with device-tuned implementations (by using CUDA or OpenCL), in many scientific applications, the performance penalty obtained by the OpenACC model is within a reasonable range (10-30%) [236]. This together with the fact that many data-parallel patterns may be expressed easily using the OpenACC model [323], makes OpenACC a first-class citizen in the context of performance portability tools and frameworks targeting GPUs for scientific computing. It is still not clear if the OpenMP and OpenACC models will converge in a single model or will co-exist for the foreseeable future.

3.3.5 The RePaRa tool-chain

The RePaRa project (Reengineering and Enabling Performance and poweR of Applications)[3] funded by the European Seventh Framework Program (EU FP7) focused on helping the transformation and deployment of new and legacy applications on parallel heterogeneous computing architectures while maintaining a balance between application performance, energy efficiency and source code maintainability.

To achieve this objective, differently from the mainstream OpenMP approach that uses compiler directives to annotate the code, in REPARA, parallelization directives are introduced as C++ attributes, which are part of the C++11 standard rather than an extension to the language.

By leveraging on C++ attributes to identify parallel regions and to introduce parallel patterns such as *pipeline*, *farm* and *Map* and their composition at higher level, the REPARA project proposed a tool-chain based on source-to-source transformations and refactoring techniques to produce parallel code targeting standard multi-cores equipped with hardware accelerators such as GPUs, FPGAs, and DSPs.

The RePaRa tool-chain is aimed at providing a methodology to parallelize a program starting from a sequential code (in some cases reshaped if necessary) by using a set of C++ attributes to define parallel regions of code called *kernels*. The methodology is structured into steps, each one taking into account different aspects of the parallelization of the code and implementing a clear *separation of concerns*. A *code annotation* phase, currently performed directly by the programmer, identifies the "kernels" subject to parallelization. Then a *source-to-source* transformation phase (built into the Cevelop IDE[4]) deals with the refactoring of the identified parallel kernels into suitable run-time calls, according to the Hardware&Software architecture(s) targeted (i.e. multi-cores equipped with GP-GPUs, FPGAs and DSPs). Finally, a

[3] RePaRa project home: `http://www.repara-project.eu/`

[4] Cevelop homepage:`https://www.cevelop.com/`

target-specific *compilation phase* generates the actual "object" (executable) code.

The first phase starts with a sequential program in which the user detects those parts of the code which can be annotated using C++ attributes. In the second phase, the annotated code is passed to the *Source-to-Source Transformation Engine*. From the annotated source code, an *Abstract Intermediate Representation* (AIR) is generated. Then, the engine uses the AIR and a set of predefined rules, specific for each parallel programming model, for determining whether the corresponding code can be transformed into a *Parallel Programming Model Specific Code* (PPMSC)[5]. The PPMSC is the parallel generated code that is functionally equivalent to the original sequential code extended with parallel kernels execution accordingly to the attribute parameters and to the selected programming model (e.g., Intel TBB and FastFlow). The third phase includes the target compilation phase using a standard C++ compiler and all low-level dependencies needed to run the code. The run-time used should provide coordination and all the mechanisms needed to support the deployment, scheduling and synchronization of kernels on the target platform(s) [112].

3.3.6 SPar

SPar is a C++ parallel programming tool providing the user with an annotation-based language aiming at modeling the main properties of stream parallel applications [177]. As in the RePaRa toolchain, SPar uses standard `ISO C++11` attributes mechanism to annotate the user's sequential code and then generate an intermediate C++ representation of the initial code containing calls to parallel run-time. SPar provides attributes to introduce parallel patterns such as *farm* and *pipeline* and `Map`. Currently, Spar supports as run-time the FastFlow framework [21].

The SPar annotation mechanism gives the application developer more flexibility than pragma annotations, which are compiler pre-processing directives which are not part of the C++ grammar. C++ annotations may be put almost anywhere in a program according to the C++ standard grammar. The standard annotation grammar is

[5] REPARA imposes some restrictions on the parallelizable source code when targeting specific hardware.

```
[[spar::ToStream, spar::Input(res,channel,src,S)]] for(;;){
 total_frames++;
 inputVideo >> src;
 if (src.empty()) break;
 [[spar::Stage, spar::Input(res,channel,src,S),
  spar::Output(res),spar::Replicate()]]{
  vector<Mat> spl;
  split(src, spl);
  for (int i =0; i < 3; ++i){
   if (i != channel) spl[i] = Mat::zeros(S, spl[0].type());
  }
  merge(spl, res);
  cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
  cv::addWeighted(res, 1.5, res, -0.5, 0, res);
  Sobel(res,res,-1,1,0,3);
 }
 [[spar::Stage, spar::Input(res)]] {
  outputVideo << res;
 }
} // for()
```

Code 8: SPar example. Code from [177].

general enough to support the customization of new attributes and determine where they are allowed in the source code (e.g., to annotate types, classes, code blocks, etc.). SPar preserves the initial sequential code, but it imposes some restrictions to ensure correct parallel code generation.

In the Code 8 is presented a snippet of code where a subset of SPar attributes are used to parallelize a video streaming computation [177]. In this example, a parallel region is identified by the `ToStream` annotation which also identifies the input data sources. Inside the parallel region, there are two `Stage` regions with the corresponding dependencies that are captured by the `Input` and `Ouput` attributes. These two stages are connected in a pipeline fashion. As the first `Stage` region may be computed independently over different input items, to increase the concurrency degree, that region is replicated by using the `Replicate` attribute. The logical computation model in this example is a two-stage pipeline where the first stage is a *Task-Farm* pattern and the second stage is sequential.

3.4 Programming distributed systems

For the sake of completeness, in this section we briefly discuss the main tools and libraries for programming distributed systems. Differently from shared-memory architectures, distributed memory systems are individual computers connected by a communication network, each having exclusive access to its private memory.

3.4.1 Message Passing Interface

The *Message Passing Interface*, commonly referred as *MPI*, is a standardization of an API implementing the message-passing programming model [298]. It defines the syntax and semantics of library routines for standard communication patterns such as *point-to-point*, *broadcast*, *scatter*, *gather*, *all-to-all* and so on. In the MPI model, the processes executed in parallel have separate memory address spaces and communications occur when two or more processes cooperate via communication primitives to explicitly transfer part of the address space of a set of processes into the local address space of other processes. Moreover, MPI implicitly follows the Single-Program Multiple-Data (SPMD) paradigm, in which all processing units execute the same program, each operating on its local chunk of data.

Many general-purpose programming languages have bindings to MPI: C, C++, Fortran, Java, and Python. Mainly targeted to distributed architectures, MPI offers specific implementations for almost any high-performance interconnection network. Likewise, implementations exist that allow us to use MPI even on standalone CMP systems (e.g., Xeon Phi).

In MPI, the workload partitioning and task mapping have to be done by the programmer, therefore the programmer has to deal with both functional and extra-functional semantics. Precisely, programmers must manage which tasks have to be computed by each process.

MPI primitives can be classified as *two-sided* and *one-sided* communication. In the first model comprising *point-to-point*, *collectives* to complete the communication both senders and receivers have to match so that some amount of synchronization is needed to manage the matching of messages. However, starting from MPI-2 [179], *one-sided* communications have been introduced to allow direct memory accesses without requiring sender/receiver matching. In this way, the data transfer and synchronization is completely decoupled. The standard provides three communication calls: *MPI_Put* (remote write), *MPI_Get* (remote read), and *MPI_Accumulate* (remote update).

Starting from MPI-2, the standard also offers *parallel I/O operations* [309] to provide access to external I/O devices exploiting complex data types and communicators (i.e. an object that connects groups of processes in an MPI session).

MPI operations can be classified depending on the local effect as viewed by the executing process and on the effect on the synchronization with other processes [279]. Considering the local view of the communicating process, the operation can be *blocking* or *non-blocking*:

- blocking: An MPI communication operation is blocking if their resources (e.g., buffers) can be reused after the function call. That means, blocking calls do not return unless the operation has completed.

- non-blocking: An MPI communication operation is non-blocking if the corresponding call may return before all effects of the operation are completed and therefore the resources associated with the operation cannot be immediately reused

Blocking and non-blocking operations performed by an MPI process do not affect the execution of other processes. Instead, synchronous and asynchronous communications have an impact on the way the communication occur among different MPI processes:

- synchronous communication: The communication between a sending process and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. This means that the completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.

- asynchronous communication: In this case, the sender can execute its communication operation without any coordination with the receiving process.

At present, the MPI standard has several versions: version 1.3 (also known as MPI-1), which emphasizes message passing and has a static run-time environment, MPI-2.2 (MPI-2), which includes new features such as parallel I/O, dynamic process management and one-sided operations, and MPI-3.1 (MPI-3), which includes extensions to the collective operations with non-blocking versions and extensions to the one-sided operations [159]. At present, the last version of the standard is MPI-4.0.

3.4.2 Partitioned Global Address Space

Writing shared-memory parallel programs is widely perceived as simpler than writing message-passing ones. In the '90s several attempts were made to build Software-based Distributed Shared Memory (SDSM) systems on top of distributed memory systems (notable examples are IVY [225], Munin [78] and ThreadMarks [205]). Though several smart distributed caching techniques have been developed to reduce costly remote memory operations, the principal obstacle for scalability of SDSM systems has been their inability to exploit locality effectively. For this reason, there has been considerable interest in developing locality-aware paradigms for shared-memory parallel programming. The `Partitioned Global Address Space` (PGAS) model was developed to address this issue. In the PGAS model, multiple SPMD processes share a global address space. However, the shared space is partitioned and a portion of it is local to each process. Programs using the PGAS model can exploit locality by having each process principally compute on data that is stored in its local memory [91].

Data structures can be allocated either globally or privately. Global data structures are distributed across address spaces of multiple nodes, typically under the control of the programmer. Remote global data are accessible to any process (or thread) as simple assignment or by using dereference operations. The compiler and run-time are responsible for converting such operations into messages between processes on a distributed memory machine.

A large number of PGAS programming systems have been implemented in the past, including Split-C [210], Unified Parallel C (UPC) [145], HabaneroUPC++ [214], Co-Array Fortran (CAF) [257], Chapel [83] and X10 [86]. Almost all these systems are implemented on top of the GASNet [56] communication library. In the UPC language, any processor can directly read and write variables on the partitioned address space, while each variable is physically associated with a single processor. Each thread is associated with a partition of the Global Address Space (GAS), which is subdivided into a local portion and a shared portion. Local data can be accessed only by the thread that owns the partition, while data in the shared portion are accessible by all threads. Threads access shared memory addresses concurrently through standard read and write instructions and using barriers and locks to synchronize the execution flow.

What is most different between general SDSM systems and PGAS ones is that in PGAS systems remote data accesses are explicit. Recently we are witnessing a renewed interest in user-level SDSM systems that trade latency with throughput by exploiting application parallelism. For example, a recent proposal called Grappa [256] provides the user with a software DSM system for commodity clusters designed for data-intensive applications. Instead of relying on locality to reduce the cost of memory accesses, Grappa exploits local-node parallelism to keep processor resources busy and hide the high cost of inter-node communication for accessing remote data.

In the context of shared memory models, another common abstraction is the so-called key-value storage which provides virtual shared spaces accessible by all the components in the system by using associative keys. Notable implementations of this model are Redis [280] and Memcached [245] mainly designed for cluster-based HPC systems.

3.4.3 Skeletons for Distributed Platforms

In this section, we briefly describe some notable implementations of high-level parallel programming based on algorithmic skeletons proposed over the last two decades. Such proposals, cover many different implementations of stream-parallel, data-parallel and task-parallel skeletons targeting distributed memory systems. For a broader presentation of algorithmic skeletons library and frameworks, please refer to the survey by González-Vélez and Leyton [170].

One of the pioneering proposals for skeleton-based parallel programming was the *Pisa Parallel Programming Language* (P^3L) developed at the Computer Science Department of the University of Pisa in the nineties [267, 268]. P^3L was a coordination language for the parallel execution of code written in C. Its language core included programming paradigms like *pipeline*, *Task-Farm* and iterative *data-parallel* skeletons. Skeletons in P^3L were used as constructs and they were the only way to express parallel computations in the coordination language. It was released with a compiler using the concept of *implementation templates* to compile the code into different target architectures [39].

The experience of P^3L later converged into the development of the *Skeleton Interface Environment* (*SkIE*) that was developed by the Computer Science Department of the University of Pisa in collaboration with QSW Ltd. [38]. *SkIE* improved P^3L in different ways: enriching the number of different languages that could be used to express different parts of the applications (C/C++/Fortran/Java) and most of all, it allowed composition of skeletons. A *SkIE* program was basically a pipeline skeleton whose stages could be a composition of *Task-Farm*s, *Map*s, *pipeline*s. The *loop* skeleton could be used only as the outermost skeleton of a more complex composition.

SKELib [117] is another fruit of the P^3L experience. It builds upon the contributions of P^3L by inheriting the template system for targeting different platforms, but it differs from P^3L because it was implemented as a C library and not as a coordination language. It provided the user with only stream-based skeletons: *Task-Farm* and *pipeline.*

ASSIST (A Software development System based upon Integrated Skeleton Technology) [317], developed starting from 2002 at the University of Pisa, was the evolution of the *SkIE* environment and a rethinking of the skeleton-based approach. The structure of an *ASSIST* program is a graph (written using the *ASSIST-CL* language), whose nodes are components and the arcs are abstract interfaces that support streams. Any graph structure is permitted. Streams are the structured way to compose modules into an application. In addition, components can also interact by means of "external" shared objects, i.e. external components not expressed in *ASSIST-CL*. Components are expressed as language modules, which may be parallel modules called *parmod* or sequential modules. A sequential module is the simplest component expressed in *ASSIST*: it has an internal state and is activated by the input stream values according to a deterministic data-flow behavior. The parallel module is able to express the semantics of several skeletons as particular cases (e.g., farm, pipeline, stencil and map) as well as to express more general parallel and distributed program structures, including both data-flow and nondeterministic reactive computations. Access to external objects is allowed inside *parmods*. The introduction of shared data structures aims to efficiently manipulate very large data sets, to simplify the programming of irregular and/or dynamic problems.

SkeTo is a C++ skeleton library based on MPI developed at University of Tokyo [238]. It mainly provides skeletons for data-parallel computation using distributed data structures, such as arrays, matrices, and trees. The semantics of the skeletons is formally given in terms of homomorphisms and homomorphism transformations according to the Bird-Meertens Formalism [49]. As the SKELib library, *SkeTo* is implemented as a library aiming at using skeletons as plain library calls within sequential C/C++ programs to accelerate the execution of a portion of code.

eSkel (the Edinburgh Skeleton Library) is a structured parallel programming library developed at the School of Informatics, University of Edinburgh. The first version of the library was developed by Murray Cole in 2002 [149]. The new version, *eSkel2* was released around 2005 [44]. *eSkel* adds skeletal programming features to the C/MPI parallel programming model. Its underlying model is SPMD, inherited from MPI, and its operations must be invoked from within a program which has already initialized an MPI environment. *eSkel* defines five skeletons: *pipeline*, *deal*, *butterfly*, *farm* and *haloswap*. The latest version available (eSkel 2.0.1) only implements *pipeline* and *deal* skeletons.

The Orléans Skeleton Library (OSL) [222] is a C++ skeleton library developed at the University of Orléans, which is based on the Bulk-Synchronous Parallel (BSP) model of parallel computation [316]. OSL provides a collection of data parallel skeletons implemented on top of MPI: *Map*, *Zip*, *Reduce*, *Scan*, *Permute*, *Shift*, *Redistribute*, *getPartition* and *Flatten*. It takes advantage of the expression templates optimisation techniques to provide good performance yet allowing programming in a functional style. OSL aims to provide an easy-to-use library which enables simple reasoning about parallel performances based on a simple and portable cost model [201].

Lithium [9] was developed in the early '00s at the University of Pisa. It was a pure Java parallel programming environment based on skeletons and has represented one of the first skeleton based programming environments implemented in Java and the first complete skeleton based Java environment exploiting macro data-flow implementation techniques [106]. One of the most interesting features of Lithium was that it used a set of code optimizations based on skeleton rewriting techniques [14].

Lithium has been reimplemented and the name changed to Muskel. Muskel only supported stateless stream parallel skeletons such as *pipeline* and *task-farms* whose run-time was still based on the macro data-flow implementation techniques. The semantics of Muskel has been provided in a formal way through labeled transition systems [15]. Muskel supports both autonomic management of non-functional feature and limited fault-tolerance. Muskel introduced the autonomic manager concept that has been later inherited by other skeleton programming frameworks. The autonomic manager of Muskel is able to ensure performance contracts concerning parallelism degree and can solve problems related to the failure of nodes used to implement the skeleton application. Muskel runs on any distributed systems supporting Java/RMI. More recently, multi-core run-time support has been introduced in the programming framework [17].

Calcium [77] provides a set of task and data parallel skeletons in Java targeting cluster of workstations. It also provides a performance tuning model which helps programmers to find performance bottlenecks within the parallel infrastructure. Skandium is a recent evolution of Calcium targeting multi-core platforms.

OcamlP3L [111, 90] has been designed in mid '90s as an Ocaml library implementation of the P^3L skeleton framework. *OcamlP3L* features the P^3L data-parallel and stream-parallel skeletons. However, differently from P^3L it was implemented as a library rather than as a new language. The functional semantics of the *OcamlP3L* skeletons is given in terms of High-Order-Functions, while their parallel semantics is given informally. *OcamlP3L* allows us to reuse the code written in Ocaml wrapped into OcamlP3L sequential skeleton. The implementation of OcamlP3L followed the template model as in P^3L. Templates may be written using POSIX processes/threads and TCP/IP sockets for targeting communications in a distributed environment. *OcamlP3L* uses functional closure for data communication across distributed Ocaml interpreters. Each interpreter running on different nodes was used to specialize the processing elements implementing the skeletons of the user application. From 2010, *OcamlP3L* is no longer maintained and a new framework called *CamlP3L* and then *Sklml* has been build using the experience of *OcamlP3L*.

Eden [226] extends the Haskell functional language by providing support for parallelism on shared and distributed memory platforms. In Eden, parallel programs are organized as a set of processes communicating via Single-Producer Single-Consumer channels. Although processes are defined explicitly, communication and synchronization issues are handled transparently to the programmer. Eden is a general-purpose parallel functional language suitable for developing sophisticated skeletons as well as for exploiting more irregular parallelism that cannot easily be captured by a predefined skeleton. Eden supports task and data parallelism through a set of skeletons that are defined on top the process abstraction layer.

To the best of our knowledge, the only C++-based skeleton-based library targeting distributed memory systems that is currently maintained is *Muesli* [212]. In *Muesli*, data-parallel skeletons are offered as member functions of distributed data structures whereas task-parallel skeletons are used to build process topologies. Data serialization, i.e. the process of transforming an object into a sequence of bytes that can be stored in a contiguous memory buffer to be sent over the network, is an important issue in C++-based distributed framework. In *Muesli* it is implemented by providing mechanisms for serialization of arbitrary data types. In particular the C++ abstract class MSL_Serializable provides the basic mechanism for serializing arbitrary data types. Each object that contains pointers to data and that has to be transmitted over the network, must be derived from this class and implement specific methods. All other objects are implicitly serialized when needed [147].

In the context of algorithmic skeletons, Behavioural Skeletons (BS) have been an interesting research effort raised within the CoreGRID programming model for the GRID platform [10, 16]. They provide stream and data parallel components equipped with autonomic managers dealing with skeletons' non-functional features (i.e. performance and security objectives). The abstract machine targeted by BS was ProActive, a distributed middleware platform for clusters of workstations and GRIDs.

3.4.4 Google's Map-Reduce

In 2004, Jeff Dean and Sanjay Ghemawat working at Google published an article in which they abstractly described large-scale map-and-reduce data processing at Google [133]. Later in 2008, an update of the original publication appeared in the ACM communications [134]. Basically, the authors proposed the so-called "Google's Map-Reduce" (GMR) as a *programming model* for processing large datasets. This was one of the few cases where a powerful and widely accepted framework has been built around a combination of only two parallel paradigms: *Map* and *Reduce*.

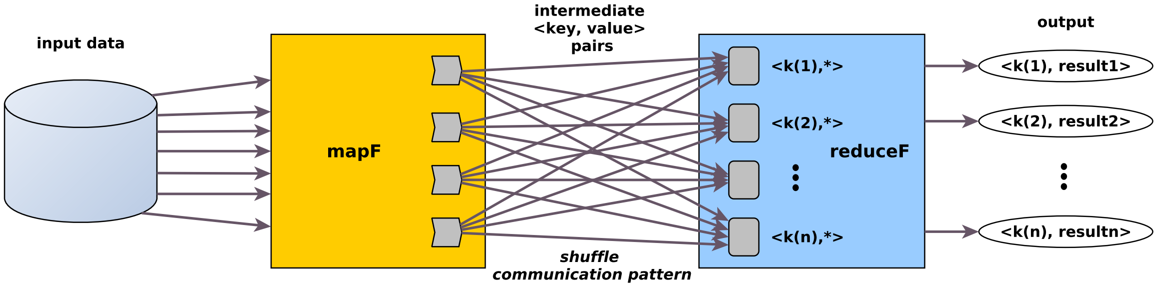


Figure 3-7: Two-phase computation schema of the Google's Map-Reduce model.

Some authors pointed out that, the GMR programming model is just a smart implementation of two well-known parallel skeletons [45, 65]. Moreover, given the great acceptance that the programming model has found, and its close relation to skeleton programming, one of the main merits of the Google's Map-Reduce model is to have brought the skeleton approach to industry [45].

The computation scheme of Google's Map-Reduce is sketched in Figure 3-7. A first instance transforms the input data, coming from a set of files of a distributed storage, into `<key,value>` pairs by using a *mapF* function such that to create collections of intermediate data having the same key value format. Then, each collection with the same key is reduced to one resulting `<key,value>` pair using a `reduceF` function. From a high-level standpoint, a GMR job can be divided into three logical macro steps: 1) a *Map* step in which each worker node applies the *mapF* function to the local data, transforming each datum into a `<key,value>` pair; the data produced is then written into a temporary storage; 2) a *shuffle* step where worker nodes redistribute data based on the value of the keys such that all data belonging to one key is located on the same worker node, and 3) a *Reduce* step where each worker computes the reduce function on local data.

The main contribution of the GMR programming model is the utilization of a key-value model for processing data and the repartitioning step (the *shuffle* step) that occurs in-between the two phases. One important aspect of this model is the data locality exploitation during the `Map` phase. The `Map` computation is performed where the data are stored, promoting the idea of moving the computation close to the data. The RTS exploits the natural data partitioning performed by the distributed file system and forcing operations to be computed using local data.

A popular open-source GMR implementation is provided by *Hadoop* [322]. The *Hadoop* framework consists of three main components: 1) a high-throughput *Hadoop Distributed File System* (HDFS), 2) a Map-Reduce engine for data processing and, 3) the *Hadoop YARN* software for resource management, job scheduling and monitoring of resources. Recently, Hadoop has been detached from the Map-Reduce processing engine, so that the HDFS+YARN software layers can be used as base tools for other frameworks, such as *Apache Spark* [332] and *Apache Storm* [223]

The Google's Map-Reduce model has also been implemented on shared-memory systems demonstrating good performance figures [278].

3.4.5 Apache Spark and Spark Streaming

Apache Spark [332] is a programming framework for large-scale data processing. It has been initially designed to overcome some limitations of Google's Map-Reduce model when it was applied to iterative computation or in large-scale interactive data-analytics. In these contexts, the cost associated with loading and storing data at each iteration on the distributed file system and the cost of data replication could be so high that the overall performance is not satisfactory. *Apache Spark* introduces an additional data layer represented by the `Resilient Distributed Dataset` (RDD) which is a read-only collection of objects partitioned across multiple processing nodes and permanently stored in the main memory of each node [330]. The RDD data layer implemented in *Spark* allows a general-purpose programming language (e.g., Scala) to be used at interactive speeds for in-memory data processing on large clusters [329].

Spark also provides a streaming library (*Spark Streaming* [331], which leverages Sparks core components to allow the definition and execution of Data Stream Processing (DSP) applications [31]. *Spark Streaming* introduced a high-level abstraction called Discretized Stream or *DStream* that basically represents a stream of RDDs with elements being the data received from real input streams. Operations over DStreams are executed at the RDD level. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees. While other specialized Stream Processing Engines (SPEs), such as for example *Apache Storm*[6], are based on a record-at-a-time processing model, i.e. input elements (called tuples in the DSP jargon) are processed as they arrive, the approach followed by Spark Streaming is based on *micro-batching*. A micro-batch is a continuous sequence of RDDs of the same type storing tuples received within a given time interval. *Spark Streaming* provides various ready to use stream operators such as *Map*, *Reduce* and *Join*. It also provides some basic operators that work on sliding-window data abstraction and multi-keyed input streams. *Spark Streaming* is one of

[6] Apache Storm homepage: `http://storm.apache.org`

the few large-scale frameworks that combine batch and continuous stream processing in a single programming model [329].

Recently, many specialized SPEs have been proposed such as *Flink* (`http://flink.apache.org`), the already mentioned *Apache Storm* and *Thrill* [48].

3.5 Summary

In this chapter, we have presented state-of-the-art frameworks and libraries for parallel programming targeting multi-cores. Also, we briefly present programming models and tools targeting distributed systems. We mainly focused on proposals promoting high-level parallel programming covering stream, task, and data-parallelism on multi-cores and GPUs. Almost all frameworks presented aim at overcoming the limitations of the traditional "*threads & locks*" low-level parallel programming model still widely used on shared-memory systems. We considered those proposals that are close to the FastFlow library approach, which is the main subject of this thesis.

The evolution of parallel computing platforms is slipping toward heterogeneous many-core systems. This evolution contributes to both blurring the boundaries between shared-memory and distributed-memory systems and to increase the complexity in parallel programming. The use of low-level APIs for programming these new emerging platforms often results in a fairly unproductive development process. Other approaches promoting code annotations such as OpenMP and OpenACC (and also the ones exploiting C++ attributes features) are becoming mainstream because of their simplicity to target both multi-cores and HW accelerators. The compiler undertakes all the necessary steps to implement parallelism guided by hints from the user (i.e. annotations). The success of these approaches is mainly due to the fact that programmers still tend to develop first a sequential version of their program or kernel and then move to a parallel version, which naturally pairs with pragma-based parallel annotations. Another important reason for using these models is that they are language-agnostic between C and Fortran, the latter being still widely used in the HPC and scientific communities. Additionally, by using sequential code annotations, the pure sequential version of the code is entirely reused. At RTS level, the significant heterogeneity of new and forthcoming systems seems to promote OpenCL has the *de facto* abstract platform. However, if on the one hand OpenCL-based RTSs deliver good performance for GPUs, and potentially even performance portability since it is efficiently implemented and offered by several vendors, on the other hand, standard multi-cores are not yet well supported by OpenCL. This is mainly due to the difficulties in mapping OpenCL abstract platform to the concrete standard multi-core platforms.

Task-based RTSs are increasingly employed on both heterogeneous multi-cores and distributed-memory systems given the great parallelism opportunities offered by the task-based programming model. In the context of distributed systems, tasks are often combined with the GAS programming model. However, the task-based parallel programming can be considered a low-level approach and it seems more suitable to implement RTSs and high-level parallel components than complex applications.

Data stream processing is emerging as a central paradigm of modern data analytics. The efficient management of streams poses several challenges in terms of latency and throughput, and recently several new frameworks have been proposed targeting multi-core systems.

Among the broad set of recent proposals, the ones based on domain-specific parallel solutions represent a viable solution to the various, heterogeneous and increasingly complex scenarios of typical multi-core architectures. In our vision, what is currently missing is a common and widely accepted software layer of well-defined, highly-efficient and portable parallel components that could be used to cover the usability needs and the performance requirements of the different application domains and also of the RTS programmers implementing the related DSLs. In this thesis, we try to fill this gap by proposing a reduced set of parallel building blocks that can be used both as RTS components for building higher-level frameworks and also as primary ingredients to increase the flexibility of the pattern-based parallel programming methodology.

Although this chapter is far from being a complete and exhaustive presentation of all existing frameworks and libraries proposed by IT companies and research institutions, we believe it provides a broad spectrum of current and past research efforts in the field of high-level parallel programming.

Chapter 4

The FastFlow parallel library

4.1 Introduction

This chapter provides an overview of the FastFlow library and its programming model. Notably, we introduce the new version of the FastFlow library by briefly presenting its new distinguishing features together with the new layered software design. During the presentation of the FastFlow components, we will provide the reader with links to next chapters where a broader discussion of the most important aspects can be found.

The FastFlow library is the result of a research effort started in 2010 by Massimo Torquati (from University of Pisa) and Marco Aldinucci (from University of Turin) with the aim of providing application designers with key features for parallel programming via suitable parallel programming abstractions and a carefully designed Run-Time System (RTS). The *structured parallel programming methodology* was the fertile ground that has allowed the development of the initial idea and then guided the FastFlow library implementation. Massimo Torquati was the developer of all the most critical and important functionalities of the FastFlow library.

The history of the FastFlow project is briefly retraced starting from its first version released in 2010 and then highlighting the most significant enhancements it had over about eight years with particular focus on the latest advancements.

4.2 Overview

FastFlow is a structured parallel programming library offering multi-level APIs to the parallel programmer. The library was conceived to support highly efficient stream parallel computations on heterogeneous multi-cores with the challenge to tackle “the 3P”: *Performance*, *Programmability*, and *Portability* (see also Section 2.4.1 for a discussion about “the 3P” criteria). The library is released open-source under the LGPLv3 licence (`http://calvados.di.unipi.it/fastflow`).

FastFlow version 1 and 2. The FastFlow library is realized as a C++ header-only template library[1] that allows the programmer to simplify the development of parallel applications modeled as a structured directed graph of processing *nodes*. The graph of concurrent nodes (also called *concurrency graph*) is constructed by the assembly of *sequential and parallel building blocks* as well as *higher-level parallel patterns* modeling typical schemas of parallel computations such as *pipeline*, *map+reduce*, *Divide&Conquer*, *stencil*, and *parallel-for*. FastFlow efficiency stems from the optimized implementation of the base communication and synchronization mechanisms and from its layered software design (see Figure 4-1).

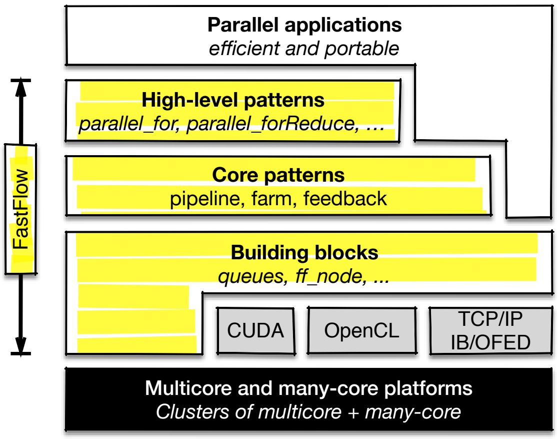


Figure 4-1: FastFlow software layers (versions 1 and 2).

In the FastFlow version 1 and 2, the bottom layer (*Building blocks* in Figure 4-1) provides the *node* and *communication channels* components. A *node* is the basic unit

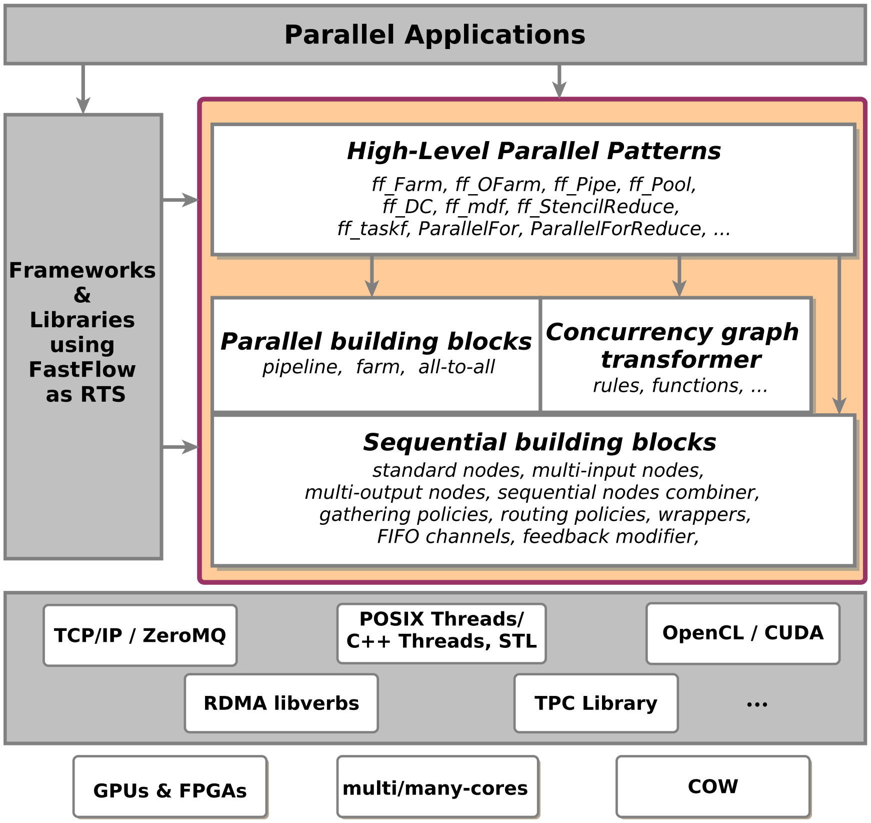
[1] Header-only libraries are compiled together with user’s code so there are more opportunities for the compiler to introduce optimizations.

of parallelism that is typically identified with a node in a data-flow process network while *communication channels* are the edges connecting nodes. A FastFlow node is used to encapsulate sequential portions of code implementing user's functions (i.e. business logic code) and also as a base abstraction of high-level parallel patterns. From the API viewpoint, a FastFlow *node* is an instance of the `ff_node` C++ class.

The second layer (called *Core patterns*) provides basic streaming parallel components (i.e. *farm* and *pipeline*). On top of core patterns, *High-level patterns* are provided to target different types of parallelism. For instance, the *ParallelFor* and the *Map* allow the user to express data parallelism in a way that is conceptually close to other popular frameworks (i.e. OpenMP and TBB). The *ParallelFor* pattern has been implemented employing the *farm* parallel component [119]. Chunks of independent loop iterations are streamed to be executed toward a pool of *farm*'s Workers. Like other parallel libraries (e.g., Intel TBB), FastFlow's *ParallelFor* pattern uses C++ lambda expression as a concise way to create function objects defining the body of the loop (the *ParallelFor* pattern is described in Chapter 9).

Some of the FastFlow high-level parallel patterns (i.e. *Map*, *Reduce* and *Stencil*) can be executed either on CPU cores or their execution can be offloaded onto GPUs. In the latter case, the user code may include GPU-specific code (i.e. CUDA or OpenCL kernels).

FastFlow version 3. In the new FastFlow version (in the following called also FastFlow-3 to distinguish it from previous versions, when necessary), the lower software layer has been redesigned so that the *building block* and the *Core pattern* layers have now been fused in a single component. Such a new low-level layer has been extended and strengthened by adding new core parallel patterns, notably the *all-to-all* and the *sequential node combiner* (discussed in Chapter 5 and Chapter 7), to enhance both the *programmability* and the overall *performance* of the library. From now on, we will refer to the new software layer simply as *building blocks.* As we shall see in Chapter 8, the FastFlow-3 version introduced also a new software component called *concurrency graph transformer* that allows us to modify and restructure the concurrency graph of FastFlow nodes to introduce optimizations and to enhance the *performance portability* of FastFlow applications. Figure 4-2 shows the new software components of the FastFlow-3 version.



Parallel Applications
Frameworks & Libraries using FastFlow as RTS
High-Level Parallel Patterns
ff_Farm, ff_OFarm, ff_Pipe, ff_Pool, ff_DC, ff_mdf, ff_StencilReduce, ff_taskf, ParallelFor, ParallelForReduce, ...
Parallel building blocks
pipeline, farm, all-to-all
Concurrency graph transformer
rules, functions, ...
Sequential building blocks
standard nodes, multi-input nodes, multi-output nodes, sequential nodes combiner, gathering policies, routing policies, wrappers, FIFO channels, feedback modifier,
TCP/IP / ZeroMQ
POSIX Threads/ C++ Threads, STL
OpenCL / CUDA
RDMA libverbs
TPC Library
...
GPUs & FPGAs
multi/many-cores
COW

Figure 4-2: New FastFlow-3 software layers.

From now on, unless otherwise noted, we will use the term FastFlow to refer to the new FastFlow-3 version.

How to use the library. According to the structured parallel programming methodology (see also Section 2.5.1), FastFlow aims at providing the application programmer with a variety of ready-to-use stream and data parallel patterns that may be easily composed and customized to implement complex parallel applications. Building blocks and high-level patterns can be used by instantiating and extending objects from the FastFlow C++ classes. Then, the entire parallel application is expressed by connecting patterns and building-blocks in a data-flow pipeline. However, if the parallel programmer instead of parallelizing the entire application needs to accelerate only some specific parts, the approach offered by the FastFlow library is about the same. From the available set of patterns, the programmer chooses the most suitable ones for accelerating the application kernel (e.g., a *Task-Farm*, a *Map*, or a *Pipeline*). Then he/she creates an instance of the pattern selected or builds a pipeline instance containing the set of patterns selected. Finally, the programmer executes the created network topology by invoking the `run` method of the external object. At some point of the main program, the method `wait` is called to wait for the termination of the parallel kernel. Alternatively, the two calls can be combined by using the synchronous `run_and_wait_end` method (see also Figure 4-3). A simple schematic example of a possible usage of the FastFlow library is sketched in Figure 4-3. In the left-hand size of the figure (Figure 4-3-a), a logical *pipeline* of four stages is created at a given point in the application code and it is executed by synchronously waiting its termination. Possibly, the results have been stored into a file or in a suitable data structure. In this case, the stream feeding the pipeline stages is generated by the first stage of the pipeline itself (e.g., reading from a file, or from a network socket) and when the stream terminates the pipeline terminates as well returning the execution control to the main thread. In the right-hand size of the same figure, FastFlow is used in the *software accelerator* mode [18], i.e. parts of the computation (i.e. kernels) are offloaded to a

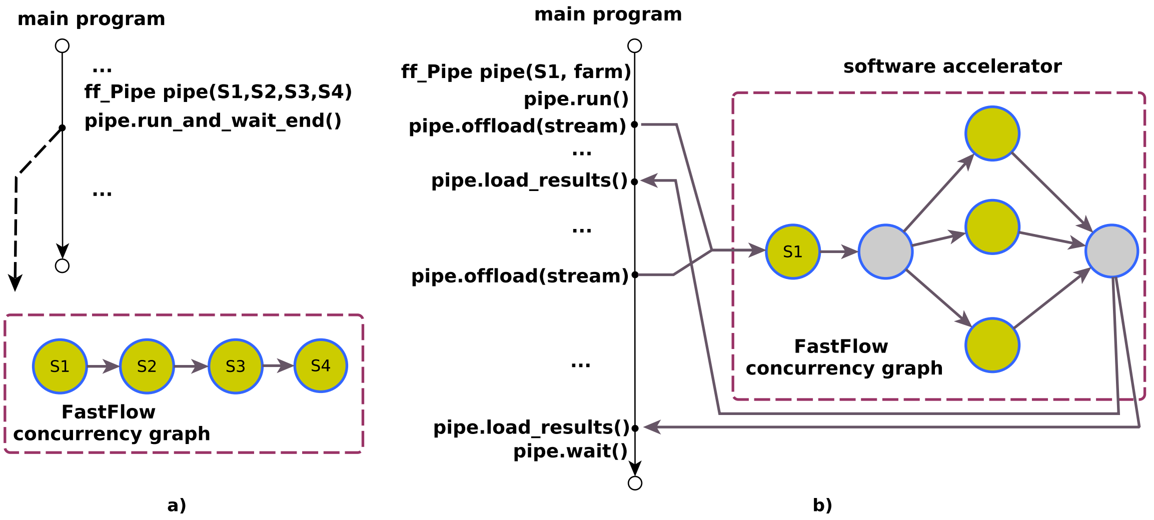


Figure 4-3: *Left:)* Standard usage of the FastFlow library: create the network topology and synchronously runs it. *Right:)* *software accelerator* mode where the FastFlow network is fed directly by the main thread.

parallel component (i.e. a pipeline containing multiple patterns), similarly to what happens when offloading computation to a hardware accelerator. The main difference compared to the previous usage scenario is that the input data stream will be provided directly by the main control flow and the output produced will be gathered by the main control flow as well.

```
struct Source: ff_node_t<float> {
    Source(std::vector<float>&D):D(D){}
    float* svc(float *) {
        for(size_t i=0;i<D.size();++i)
            ff_send_out(&D[i]);  // streaming elements
        return EOS; // End-Of-Stream
    }
    std::vector<float>& D;
};
struct Worker: ff_node_t<float> {
    float* svc(float* in) { return F(*in);  }
};
struct Sink: ff_node_t<float> {
    float* svc(float *in) { sum +=*in; return GO_ON; }
    float sum = 0.0;
};
int main(int argc, char *argv[]) {
    ...
    Source  S(D);    // Source object
    std::vector<ff_node*> W;  // pool of 4 Workers
    for(size_t i=0;i<4;++i) W.push_back(new Worker());
    Sink  R;      // Sink object
    ff_farm farm(W,S,R);   // farm building block
    farm.run_and_wait_end(); // run and wait for termination
    ...
}
```

Code 9: Parallelization with FastFlow building blocks of a simple kernel computing $\sum_i^N F(D[i])$ where D is an input vector of size N and F is a given user function.

Simple example. To show how a FastFlow code looks like, a basic example is sketched in Code 9. The *farm* building block is used to compute the simple operation $\sum_i^N F(D[i])$ where D is a vector of `floats` of size N. This simple kernel can be parallelized in different ways. We want to show here first how the `farm` building block and the concept of *data streaming* can be used for the parallelization of this simple use-case, then we show how to implement the same code by using a high-level parallel pattern (the *ParallelForReduce*). The first approach allows the programmer to have full control over the parallelization with lots of opportunity for customization, the second approach requires less programming effort hiding almost all low-level details of the parallelization.

```
int main(int argc, char *argv[]) {
   ...
  ParallelForReduce<float> pfr; // creating the pattern
  float sum{0.0}; // reduction variable
  pfr.parallel_reduce(sum,0,0,D.size(),
     // map+reduce function
     [&](const long i, float& sum) { sum+=Fun(i); },
     [](float& v, const float& e) { v+=e;}, // reduce function
     4); // parallelism degree
   ...
}
```

Code 10: High-level version of the same code presented in Code 9 that uses the FastFlow `ParallelForReduce` pattern.

All data elements of the array D are streamed toward the pool of farm Workers by the farm Emitter (in the example only 4 Workers are used). The reduction phase (i.e. the summation of all results) is computed sequentially by the farm Collector and in pipeline with the Emitter and the pool of Workers. Basically the *farm* building block is semantically equivalent to a three-stage pipeline whose middle stage is replicated a given number of times. The first node, which produces the stream of data, is defined at line 1 and it is instantiated at line 19. It generates a stream of N elements (N=`D.size()`) by using the method `ff_send_out` (line 5) and then it generates the *EOS* special value (*End-Of-Stream*), which allows the RTS to start the termination phase. From line 20 to line 21 an STL vector containing 4 replicas of the Worker node is created. The Worker is defined at line 10. Finally, the `Sink` node is defined at line 13 and instantiated at line 22. It collects all results produced by farm Workers and accumulates partial results in the `sum` local variable. The farm object uses as

Emitter the Source node and as Collector the Sink node. It is created at line 23 by passing as arguments of the `ff_farm` constructor the vector containing the Worker replicas and the objects implementing the Source and Sink nodes.

The previous simple example is clearly a *map+reduce* computation, and it could be conveniently parallelized by using the FastFlow *ParallelForReduce* high-level pattern. In Code 10 is sketched the parallelization of the same kernel described before but in this case implemented by using the `ParallelForReduce` parallel pattern. A detailed description of the `ParallelForReduce` pattern will be provided in Chapter 9.

Communication channels. FastFlow provides a fast implementation of bounded and unbounded lock-free Single-Producer Single-Consumer (SPSC) FIFO channel [19] carrying pointers to data allocated into the shared address space (see Chapter 6 for an in-depth discussion about FastFlow channels).

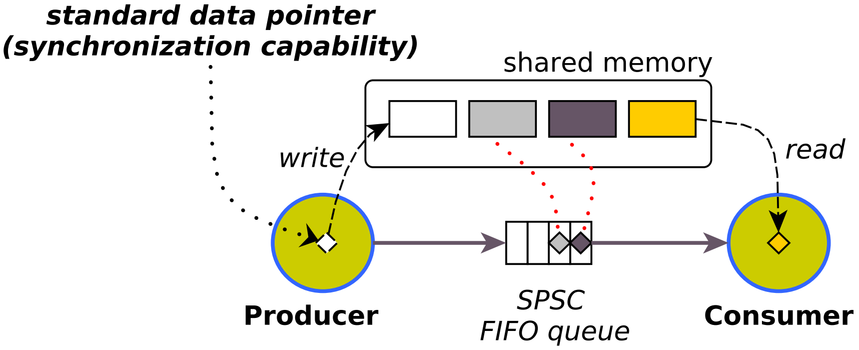


Figure 4-4: Sending references to shared data over a SPSC FIFO channel.

The FastFlow semantics of sending references over a channel is that of transferring the ownership of the value pointed from the sender node to the receiver node (see also the schema in Figure 4-4). According to this semantics, the receiver is expected to have exclusive access to the data value. Communication channels are one of the distinguishing features at the building block layer. Multi-Producer Single-Consumer (MPSC), Single-Producer Multi-Consumer (SPMC) and Multi-Producer Multi-Consumer (MPMC) interactions are not realized as concurrent passive data structures (i.e. concurrent queues). Instead, they are implemented by using mediator nodes to avoid both any memory fence induced by atomic operations (e.g., Compare-

And-Swap) and to decouple data consistency from concurrency management. The mediator nodes (for example the Emitter and the Collector nodes in a *farm* building block) use multiple lock-free SPSC queues to distribute and/or collect data elements to and from other computing or mediator nodes. They read data pointers from one or more SPSC queues and write data pointers into one or more SPSC queues and these operations use only loads and stores instructions. This approach not only avoids using atomic operations (SPSC queues can be implemented without using any memory fences on a TSO memory model and by using just one *Write Memory Barrier* on relaxed memory models [165]), but also allows us to employ mediator nodes for implementing real data processing on input data according to the data-flow model of computation. For example, in the farm building block, the Emitter and Collector node can execute user-defined code on each input item, thus combining data management with computation. Moreover, by using independent queues for each input/output channel, it is possible to have full control of data movement, for example between a producer node and a set of consumer nodes. The performance advantage of this solution for implementing streaming networks comes mainly from the higher speed of memory operations compared to atomic operations, and to the possibility to reduce cache-coherence memory traffic when using a producer-consumer model, which, in turn, provides a better support for fine-grained computations.

Building blocks. The FastFlow building blocks are concurrent components that are the fundamental elements of any structured parallel applications implemented using the FastFlow library. Building blocks are either *sequential* or *parallel*. Sequential building blocks are FastFlow nodes with one or more input or output channels. Parallel building blocks are concurrent components made out of a proper assembly of multiple nodes and multiple SPSC FIFO channels. We identified three parallel building blocks (two of them were included in the previous FastFlow versions as core patterns), *pipeline*, *farm* and *all-to-all* (described in Chapter 5), which can be specialized in different ways using also the *feedback channel* modifier (see Figure 4-5). The *pipeline* is used both for connecting building blocks and to express data-flow pipeline parallelism at run-time. The *farm* models functional replications coordi-

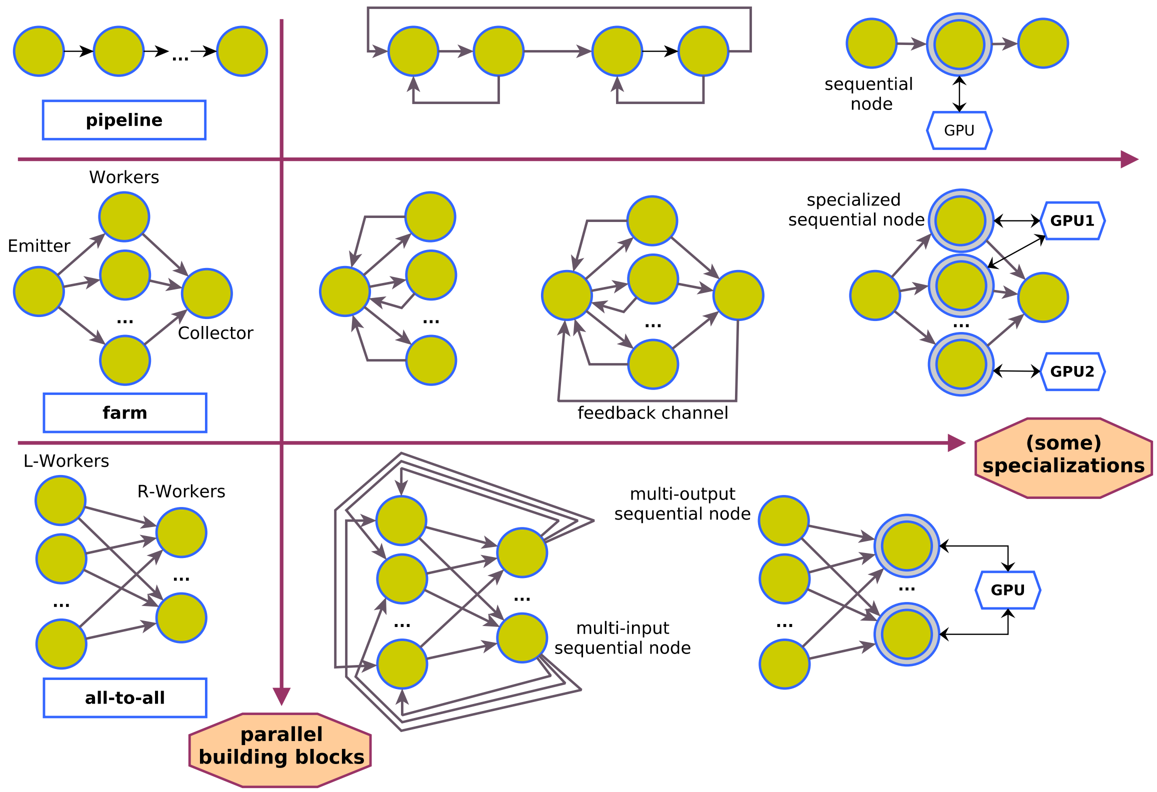


Figure 4-5: FastFlow parallel building blocks and some possible specializations of them.

nated by a centralized Emitter entity and a centralized Collector entity (that might not be present), which can be specialized by the user to define custom data distribution and data gathering policies. The *all-to-all* building block models both functional replication without a centralized coordination entity as well as the *shuffle communication pattern* between function replicas. It allows to remove potential bottlenecks in the topology introduced by the *farm* building block having one or two centralized entities (i.e. the Emitter and the Collector). The *all-to-all* also enables the *fusion* operation of two (or more) *farms* in pipeline. Concurrency graph transformations are discussed in Chapter 8. From the programmer perspective, the reduced set of sequential and parallel building blocks with their customizability and composability features, enables the so-called *LEGO-style* approach to parallel programming where the "bricks" can be either complex pre-assembled and already tested structures or elementary sequential and parallel building blocks.

The FastFlow *building blocks* motivations and characteristics are presented in

Chapter 5. The details of their implementation and performance assessments are presented in Chapter 7.

Parallel patterns. Parallel patterns are schemas of parallel computations that recur in many algorithms and applications. For each pattern, there exist different parametric implementations for a given class of target platforms, e.g., multi-cores and distributed-memory systems. They can be used in many different contexts and are targeted both to the parallelization of sequential legacy code and to the development of brand-new parallel applications. Some patterns have the additional important feature of being composable. For example, *Pipeline* and *Task-Farm* patterns can be composed and nested in almost all possible combinations. Often, parallel patterns are equipped with self-optimization capabilities (e.g., load-balancing policies, grain and parallelism-degree auto-tuning). Moreover, some patterns may target multiple devices, for example, the *Map* one can have both a CPU-based implementation as well as a GPU-based implementation.

In FastFlow, all parallel patterns available are implemented on top of sequential and parallel building blocks. They are parametric implementations of well-known structures suitable for parallelism exploitation. The high-level patterns currently available in FastFlow library are summarized in Table 4.1. A notable subset of them will be described and evaluated in Chapter 9 (i.e. the *ParallelFor*, the *ParallelForReduce*, the *Macro Data-Flow* and the *Divide & Conquer*). The FastFlow parallel patterns layer is in continuous evolution. As soon as new patterns are recognized or new smart implementations are available for the existing patterns, they are added to the high-level layer and provided to the user.

4.3 Programming model

The programming model offered by the FastFlow library is inspired by the well-known Data-Flow parallel model (see Section 2.4.4 for a concise introduction of the Data-Flow model). A FastFlow program can be built through the specialization and pipeline composition of sequential and parallel building blocks as well as high-level parallel

FastFlow Name	**Pattern description**
`ff_Pipe`	Pipeline pattern modeling a data-flow *sequence* of sequential and parallel patterns. It allows to implement both linear and non-linear compositions of parallel patterns. The FastFlow implementation of this pattern is based on the *pipeline* building block.
`ff_Farm`, `ff_OFarm`	Task-farm pattern modeling functional replication. The `ff_OFarm` pattern guarantees to preserve input ordering. Both are built on top of the *farm* building block.
`ff_Map`	In this pattern a function F is applied to all elements of the input collection. The parallel calculation of the function F over disjoint partitions of the initial collection does not require any communication/synchronization except for a barrier at the end. It is implemented on top of the *farm* building block by using the `broadcast` and `gather_all` distribution and collection policies. The FastFlow library provides also an implementation based on the `ParallelForReduce` pattern.
`ff_mdf`	It models the Macro Data-Flow execution model. A program is interpreted by focusing on the functional dependencies among data [20, 69].
`poolEvolution`	This pattern models the evolution of a population according to the principles typical of evolutionary computing [12].
`ff_DC`	This pattern models *Divide & Conquer* computations [109].
`ParallelFor`, `ParallelForReduce`	They model data-parallel computations, allowing the parallelization of loops with independent iterations and also having reduction variables [119].
`ff_stencilReduceOCL`, `ff_stencilReduceCUDA`	These patterns model iterative stencil plus reduce computations targeting GPU accelerators either using OpenCL or using CUDA code [23, 8].

Table 4.1: High-level parallel patterns currently provided by the FastFlow library.

patterns. All components of the pipeline are created by extending proper FastFlow interfaces. No centralized entity coordinating the execution of building blocks and patterns instances is present. As discussed in the previous section, building blocks and parallel patterns are structured collections of *nodes* connected by SPSC channels.

A *node* is a stateful concurrent object that is triggered as soon as an input message is present in one of its input channels. If the *node* has more than one input channel, the choice of which channel will be selected for checking the presence of input messages, is *non-deterministic*. Once a channel is selected, a round-robin polling strategy guarantees *fairness* when multiple messages are present into distinct channels at the same moment.

A FastFlow *node* represents a basic unit of computation. Each *node* can have zero or more input channels and zero or more output channels. For each input message, a *node* can send zero, one or more output messages in its output channels. Messages are removed from input channels upon reading. As soon as a message is received, the service function of the node (i.e. the `svc` method of the *ff_node* class) is invoked by the RTS passing as argument a pointer to the input message just received. FastFlow *nodes* with no input channel are activated by the RTS passing a `nullptr` argument to the service function. Message exchange between nodes is asynchronous. The capacity of a channel connecting two nodes can be either *bounded* to a fixed value or *unbounded* (the channel size is set when the parallel component is instantiated). In the first case, if the output channel is full, the node will be blocked until there is free space in the channel. Feedback channels, i.e. graph's edges directed in the opposite direction than the standard data flow, always have an *unbounded capacity* to avoid deadlock situations (see Chapter 6 for the implementation of channels and Section 7.3.1 for a discussion about potential deadlock situations).

In the following we discuss some important aspects characterizing the FastFlow programming model.

Program Termination. A *node* terminates if it receives an `EOS` message in input or if the `EOS` value is returned from its service function. The `EOS` message is a special message defined in the FastFlow namespace (there are a few other special messages that will be introduced in Chapter 7). `EOS` messages are propagated by all *nodes* in a pipeline fashion producing their termination and eventually the termination of the application. *Nodes* with multiple input channels terminate and propagate the `EOS` value only if they receive an `EOS` message from all input channels. *Nodes* with multiple output channels, propagate the EOS message into all output channels and then they terminate. The termination of programs whose topologies have at least one cycle is more complex and it is not automatic. It necessitates extra control code provided by the user. To facilitate the implementation of the termination protocol, the ff_node class provides the eosnotify method. This method is called by the RTS as soon an EOS message is received from one input channel, providing also the id of the channel. By using this method, it is straightforward to implement a termination protocol and to propagate EOS messages properly. A discussion about program termination for FastFlow concurrency graphs containing cycles, is provided in Section 7.3.1.

State management. Each *node*, upon activation, can read and modify its internal state. *Nodes* can also read and modify global states. In this case, it is the programmer's responsibility is to protect conflicting accesses to the global state. Though the model does not prevent the use of locking for protecting shared global states, the FastFlow model promotes a different approach for dealing with shared states. Whenever possible, concurrent accesses to a global state should be modeled through parallel building blocks and the state either *partitioned* or *replicated* among available *nodes* belonging to the building block. Communication channels are then used as synchronization mechanisms for coordinating accesses to the shared state. This means that, when a generic node (i.e. a sequential building block or a *node* being part of a parallel building block) receives a data pointer to the global data structure, it is intended that it has exclusive access to the data structure (either a partition or the entire data structure). When the *node* has finished working on the data structure, it passes the pointer (i.e. the *capability* of accessing the shared state) to another *node*. If the state is neither partitionable nor replicable, this approach may lead to using mediator nodes (also called "service nodes") hosting the entire data structure and so with exclusive accesses to it. Recently, this model of computation has been carefully investigated on multi-core platforms and has demonstrated a significant performance advantage against coarse-grained locking approaches [283]. In addition to that, the Parallel Computing research group at the University of Pisa is currently investigating opportunities for defining suitable abstraction to simplify state management, i.e.

state access patterns. Such patterns can be directly instantiated by the application programmer within a FastFlow network, counting on a clear functional and parallel semantics and performance guarantees [113].

Input non-determinism. FastFlow *nodes* with multiple input channels receive data elements in a non-deterministic way. This aspect, together with the possibility to access global state, represent the two major points of difference with respect to the pure Data-Flow model. However, the management of input non-determinism is of foremost importance to increase performance in data-flow networks. An example is the well-known *Task-Farm* pattern where the scheduling policy can be either dynamic (e.g., *on-demand*) or deterministic (e.g., *round-robin*). In many real use-cases, the first policy provides superior performance thanks to a better workload balancing among the farm's Workers. Input non-determinism and asynchronous message exchange among nodes are two distinguishing features of the FastFlow model.

Parallelism exploitation. In a Data-Flow model, parallelism comes from the execution of nodes with no direct dependencies. Besides, to increase parallelism, some nodes can be replicated to let them work on disjoint partitions of the same input data. This operation may require to add extra service nodes, which do not execute business logic, needed to perform the split of input data and for the aggregation of the partial results. Instead, in the case of nodes with dependencies, parallelism can still be exploited by *data streaming*, hence by the concurrent execution of multiple subsequent nodes in the concurrency graph. In the FastFlow programming model, parallelism is obtained both through the functional composition of sequential and parallel building blocks as well as by exploiting stream parallelism. Data streams can either be native (i.e. generated by external sources – *eso-stream*), or can be auto-generated by one of the stage of the network (*endo-stream*), for example by reading files or by splitting/partitioning a data structure. Data-parallelism and task-parallelism are then utilized *within* specific parallel components of the data-flow concurrency graph. For example, the *Map* and *ParallelFor* patterns are used for exploiting data-parallelism while the *Macro Data-Flow* and the *Divide&Conquer* patterns are used for exploiting task-parallelism.

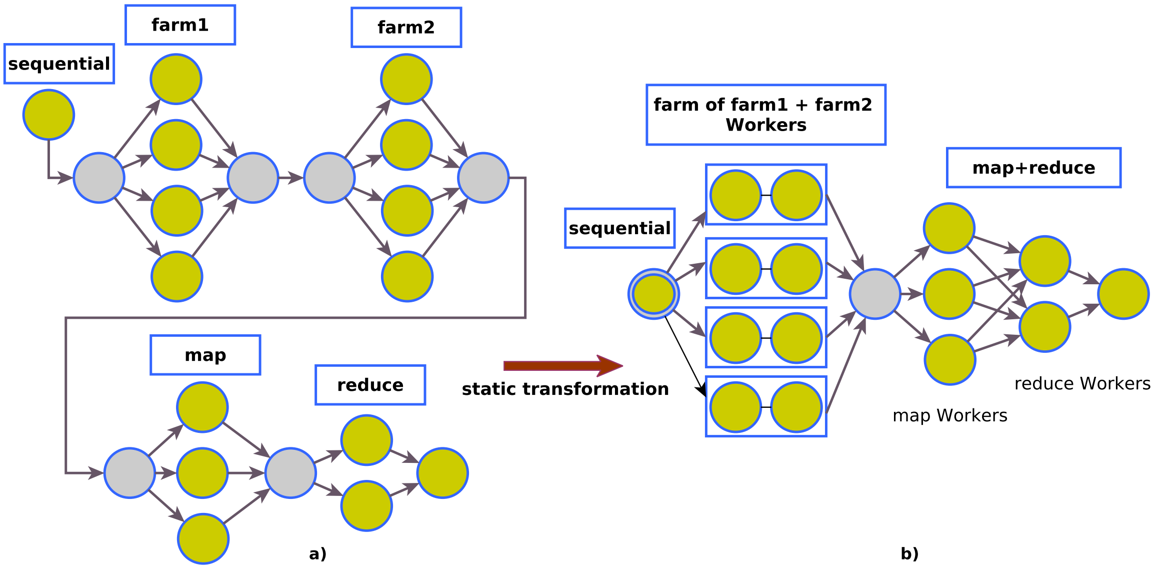


Figure 4-6: An example of a possible building blocks fusion to reduce the number of nodes. Initial graph structure (a), optimized graph structure (b).

Process network. Eventually, the flattened FastFlow application graph is composed of sequential nodes connected by point-to-point FIFO channels. Therefore, the directed graph describing the data-flow network can be analyzed and possibly statically transformed/optimized to reduce the number of nodes in the graph, still preserving the original functional semantics, with the aim of improving resource efficiency and performance. Optimizations can be applied, for example, by merging sequential nodes (e.g., those with low service times), or by applying the so-called *normal-form* [14] or by fusing farms building block implementing *Map* and *Reduce* patterns (i.e. *farm fusion* and *farm combine*). As an example, on the left-hand side of Figure 4-6 is sketched a FastFlow graph composed by a pipeline composition of sequential and parallel building blocks where nodes are stateless. This sample topology can be automatically reduced to an equivalent one as sketched on the right-hand side of Figure 4-6. In this case, the farm's Workers have been combined. Likewise, *Map*'s and *Reduce*'s Workers are used as L-Worker and R-Worker of an *all-to-all* building block (see also Figure 4-5). In the context of pure functional polymorphic functions, Wadler's "Theorems for free" [321] provides a solid theoretical background for studying interesting transformations preserving functional equivalence. Concurrency graph transformations, and some notable static optimizations are discussed in Chapter 8.

Deployment of FastFlow programs. The FastFlow concurrency graph resulting from the transformation/optimization phase, can be directly deployed onto the parallel platform. The FastFlow programming model does not use any intermediate scheduler to execute the graph; instead, it relies on the straight deployment of the entire process network on the target architecture. In FastFlow, nodes are implemented with "heavy-weight" threads of control and not with light-weight concurrent abstractions as happens in several Actor-based implementations (e.g., the CAF framework [87]). The primary motivations for this decision are as follows:

(i) The experiences in developing complex parallel applications matured within the structured parallel programming community over the last thirty years, report that typically a structured parallel application is composed by a small number of patterns (or pattern compositions) where some of them can have a large parallelism degree (for example a *Task-Farm* pattern having many Workers). Recently, we have demonstrated that a small set of parallel patterns (all implemented by the FastFlow library either as high-level patterns or as building blocks) are sufficient to efficiently parallelize the PARSEC benchmarks [129]. Other experiences in this respect, include the one developed in the context of the *Delite* parallel framework (see also Section 3.2.1), where a large number of DSLs covering different application domains, have been implemented by using a limited set of well-known parallel patterns [276].

(ii) Very fine-grained stream parallelism necessitates full control of threads on the target platforms to be able to use available cores at maximum utilization. As we will discuss in Chapter 6, FastFlow channels exhibit a latency down to a few tens of nanoseconds per message on state-of-the-art multi-cores. This allows us to build pipeline networks sustaining very high throughput, and this is only possible by leveraging low-level mechanisms at the level of threads, such as *thread-to-core affinity* and cache-to-cache fast communications [165, 19, 67].

(iii) Decoupling high-level parallel patterns from lower-level flexible and customizable building blocks helps to address both the RTS programmer and the application programmer needs. The former will mainly use building blocks for creating new patterns or new frameworks with sophisticated policies addressing specific application domains. The latter will use high-level patterns instead, which provide more specific and ready-to-use solutions to a given problem. From the RTS programmer standpoint, the use of a structured approach to parallel programming with a few and well-defined building blocks with clear functional and parallel semantics and a clear deployment process, allows him/her to use a programming model without any further software layer, thus with no surprise at execution time. Moreover, the approach based on building blocks facilitates application debugging that is one of the most critical aspect of parallel software development.

(iv) The targeting of multiple HW accelerators (e.g., GPUs, FPGAs, and DSPs) is easier if the single node is an independent concurrent entity. In fact, it is possible to specialize nodes for different target devices to manage data-movement and device kernel execution. In this respect, the FastFlow programming model facilitates the integration with other parallel frameworks, too. For example, it is straightforward to encapsulate SkePU code (see Section 3.1.1) within a FastFlow node far managing one or multiple GPUs using the SkePU containers.

To conclude, being already widely recognized that the *Programmability* challenge can be addressed only by means of high-level parallel abstractions such as parallel patterns, we do believe that the Data-Flow-like parallel programming model offered by the FastFlow library with its multi-level API and distinctive features is a good compromise between absolute performance figures, multi-core scalability, portability and performance portability targeting. Examples of existing frameworks leveraging the FastFlow programming model are Nornir [293], SPar [177] and PiCo [251].

4.4 New FastFlow features

In this section, we describe the *new features* of the FastFlow version 3. Before enumerating the main differences with previous versions, it is worth remarking that the author of this thesis developed almost all low- and high-level features of the FastFlow library since its early versions. Specifically, the communication channels and initial building blocks, the core patterns, the distributed version, the memory allocator, the OpenCL-based GPU support, and up to several high-level parallel patterns. Some of these features that we developed, will not be discussed within this thesis. Anyhow, we will provide references for the interested reader in Section 4.5.1.

The new FastFlow version presented in this thesis added several new features and functionalities to the library and they constitute the main contributions of this dissertation. The new version can be downloaded from the GitHub public repository using the following command:

Download: `git clone https://github.com/fastflow/fastflow.git`

In the following, we enumerate the main differences of the new version compared to the previous releases.

1. The basic software layer has been completely redesigned so that the *building blocks* layer and the *Core pattern layer* of the previous versions (see Figure 4-1) have been fused in a single layer called *building blocks*. This layer has been also extended by introducing two new components (the *all-to-all* parallel building block and the *sequential node combiner*), which allow us to further increase the expressive power and the flexibility of the FastFlow library.

2. The new *all-to-all* building block allows us: i) to eliminate potential bottlenecks introduced by the Emitter and Collector entities of the *farm* building block; ii) to redesign farm-based parallel patterns by using a completely decentralized configuration, therefore adding a new dimension in the software design space; iii) to enable the introduction of automatic concurrency graph transformations, such as for example the *farm combine* operation which merges two *farm* building

blocks in pipeline.

3. The new *sequential nodes combiner* building block, supporting the merging of sequential nodes, allows us to partially decouple the logical structure of the FastFlow network from its concrete deployment on the target platform without changing the FastFlow RTS. Moreover, it increases the composability of sequential nodes *and* the composability of sequential and parallel building blocks.

4. The implementation of the *multi-input* and *multi-output* nodes have been redesigned to improve their capability to connect to other nodes by removing several limitations present in the previous versions. The input and output channel cardinality of *multi-input* and *multi-output* nodes are defined only when they will be connected to another sequential or parallel building block. The "lazy" evaluation of their degree of connectivity allowed us to connect them to other building blocks greedily.

5. A new layer for implementing automatic concurrency graph transformations has been introduced (the "Concurrency graph transformer" in Figure 4-2). By leveraging a set of helper functions that allows us to combine two or more building blocks (both sequential as well as parallel) with a clear parallel semantics, the new layer collects all potential optimizations that can be applied to the network topology to reduce either resource utilization (i.e. the number of RTS threads) or to remove potential bottlenecks. A subset of transformations implemented so far are: i) removing all mediator nodes that are not explicitly implemented by the user (i.e. default *farm*'s Emitter and Collector nodes); ii) combining a sequence of *farm*s with the same number of Workers (*farm fusion* transformation); iii) merging two farms with different numbers of Workers by using the *all-to-all* building block (*farm combine* transformation); iv) automatically enabling the blocking concurrency control mode if the number of threads is greater than a given threshold (typically the number of physical cores); v) controlling the mapping of threads on available cores.

6. The implementation of both *blocking and non-blocking concurrency control mech-*

anisms for accessing the communication channels connecting two FastFlow nodes (see Chapter 6). The proposed mechanisms allow us to improve power saving and/or throughput by statically and dynamically switching the concurrency mode between passive waiting (blocking) and active-waiting (non-blocking) of run-time threads. This is particularly relevant for long-running data streaming computations where variable arrival rates and sudden workload changes require different levels of reactiveness to respect a given QoS level.

7. A new implementation of concurrency throttling mechanisms in the *farm* building blocks. This enables the development of sophisticated policies that dynamically change the concurrency level of parts of the parallel application (implemented through the *farm* building block) to increase either the sustained throughput or to reduce the power consumption by decreasing/increasing the number of Worker nodes.

4.5 FastFlow project history

The FastFlow project started in 2010 after some preliminary discussions and ideas we had with Prof. Marco Aldinucci from the University of Turin. Over the years several other people (mainly from the Parallel Computing Group of the University of Pisa and Turin) contributed with ideas and code to the development of the project. Many people around the world used FastFlow either for research purposes and industrial software development. Since the very beginning of the project, we have been (and still we are) the maintainer and the leading developer of the FastFlow library.

The first version of FastFlow was released at the beginning of 2010. Figure 4-7 summarize the timeline of the most relevant releases and the relationship with the three EU-funded research projects (ParaPhrase, RePaRa, and RePhrase) that allowed us to enhance the FastFlow library and to contribute to the success of the projects.

Given the importance of the three EU-funded research projects in the history of the FastFlow project development, we provide a brief description of each of them highlighting the most significant enhancement developed in the FastFlow library.

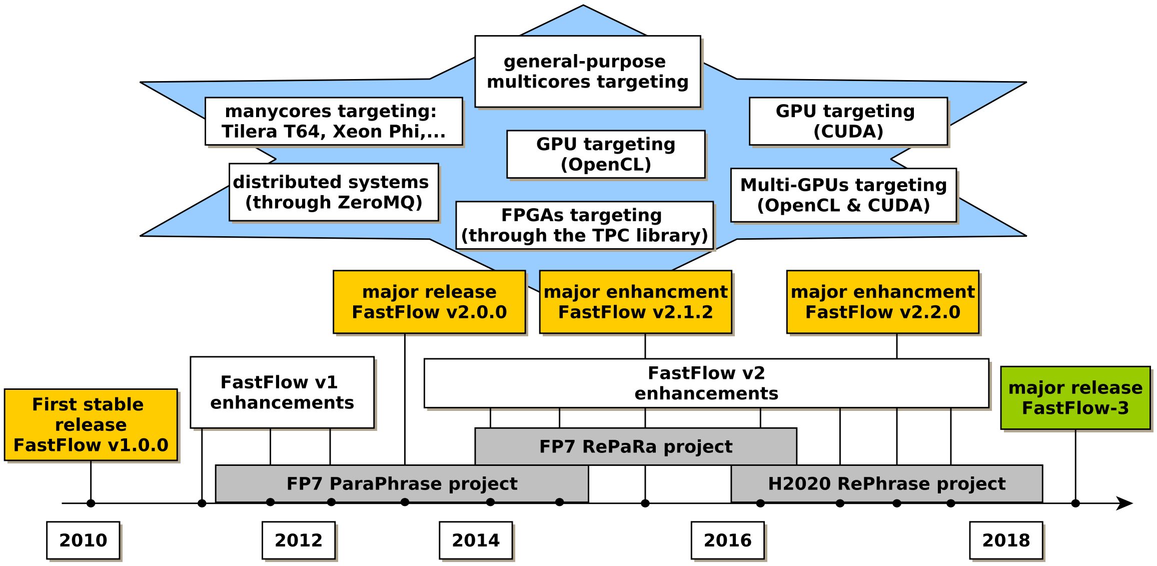

general-purpose multicores targeting
manycores targeting: Tilera T64, Xeon Phi,...
GPU targeting (CUDA)
GPU targeting (OpenCL)
distributed systems (through ZeroMQ)
Multi-GPUs targeting (OpenCL & CUDA)
FPGAs targeting (through the TPC library)
major release FastFlow v2.0.0
major enhancment FastFlow v2.1.2
major enhancment FastFlow v2.2.0
FastFlow v1 enhancements
FastFlow v2 enhancements
major release FastFlow-3
First stable release FastFlow v1.0.0
FP7 RePaRa project
FP7 ParaPhrase project
H2020 RePhrase project
2010
2012
2014
2016
2018

Figure 4-7: FastFlow project history.

The RePhrase project. The RePhrase H2020 project (Refactoring Parallel Heterogeneous Resource-Aware Applications a Software Engineering Approach)[2] started in April 2015 and lasted three years. The focus of the RePhrase project was on producing new software engineering tools, techniques and methodologies for developing data-intensive applications in C++, targeting heterogeneous multi-core systems combining CPUs and GPUs into a coherent parallel platform.

FastFlow was one of the RTSs used in the project together with OpenMP and Intel TBB. A uniform high-level interface for all RTSs is provided by GrPPI (see Section 3.1.5). The FastFlow library provided the implementation of all parallel patterns identified within the project. In the project time span, the FastFlow library has been extended mainly with *Data Stream Processing* (DSP) parallel patterns such as *Window Farming* and *Keyed Farm* [231] and the internal tracing of resource usage has been improved. All FastFlow parallel patterns used within the RePhrase projects have been summarized and assessed in Danelutto et al. [107]. Within the RePhrase project, the *RPL-sh* shell has also been developed. Is is a *skeleton*-based DSL for design-space exploration implemented on top of the FastFlow library [163]. The tool allows exploring the space of functionally equivalent but alternatives implementations

[2] RePhrase project home: `http://rephrase-eu.weebly.com/`

of the same parallel application with different non-functional properties.

The RePaRa project. The RePaRa project (Reengineering and Enabling Performance and poweR of Applications)[3] was an EU FP7 project aiming to deploy software kernels of a sequential C++ applications targeting heterogeneous multi-cores by using static or dynamic scheduling and mapping techniques and maintaining a balance between performance, energy efficiency and source code maintainability. The project started in September 2013 and lasted three years.

The FastFlow library was the RTS for the RePaRa tool-chain (see also Section 3.3.5). The library has been significantly extended to support parallel execution of RePaRa kernels (identified by using attributes code annotation introduced in modern C++) on GPU devices by using OpenCL, on FPGA devices by using the *ThreadPoolComposer* (TPC) library [209] and on both standard multi-cores and DSP systems by using standard POSIX threads [237]. A description of the RePara C++ attributes and how the FastFlow library has been used within the project is described in Danelutto et al. [112].

The ParaPhrase project. The ParaPhrase project (Parallel Patterns for Adaptive Heterogeneous Multicore Systems) started in October 2011 and lasted 42 months[4]. It aimed at producing a new structured design and implementation process for heterogeneous parallel architectures. Software developers using the ParaPhrase tool-chain may use a set of parallel patterns to develop component based applications targeting heterogeneous multi-cores. The key objectives of the project were both *programmability* by reducing time-to-solution, and better resource utilization of emerging parallel heterogeneous CPU/GPU architectures.

The FastFlow library was used as RTS for C++ programming (while Erlang was used for functional programming). During the project, FastFlow version 1 was stabilized, the first RTS support for GPUs (targeting both CUDA and OpenCL kernels) was introduced and the first implementation of the *ParallelFor* [119] and *Macro Data-Flow* [20] parallel patterns were introduced in the library.

[3] RePaRa project home: `http://www.repara-project.eu/`

[4] ParaPhrase project home: `http://paraphrase.comp.rgu.ac.uk/The_ParaPhrase_Project`

A summary of the main objectives and results achieved during the ParaPhrase project can be found in Hammond et al. [187].

Research tools leveraging the FastFlow library. Over the years several open-source research tools have been developed on top of the FastFlow library. Among these, some have been developed within Ph.D. thesis and are still maintained: SPar [177], Nornir [293], PiCo [251], and PEI [168]. Other frameworks in different application domains employed FastFlow patterns such as YaDT [24] and Peafowl [110] and PWHAT-SHUP [60]. Currently, the *WindFlow* parallel library (under active development at the time of writing), which targets high-performance Data Stream Processing (DSP) applications, is using FastFlow building blocks and its nesting features for the development of DSP parallel patterns[5]. The GrPPI [135] library (see also Section 3.1.5), developed at University Carlos III de Madrid, recently has added FastFlow as one of the possible backends.

Finally, FastFlow has also been used and tested in several industrial settings for internal testing and products developments.

FastFlow **used for teaching Parallel Programming.** Starting from AY 2011-2012, the FastFlow library was introduced as one of the parallel programming tools within the course of "System Paradigms and Models" held by Prof. Marco Danelutto for the *Master Degree in Computer Science and Networking* at University of Pisa. In the course, several parallel programming frameworks are presented to discuss different aspects of parallel programming. The final exam project assignment has to be implemented by the students with the FastFlow library. The main reason of this choice is that it provides both powerful parallel patterns and also efficient building blocks that can be used by the students to solve the critical aspect of the project assigned following the structured parallel programming methodology taught in the course. A summary of the experiences over several years of teaching structured parallel programming is reported in Danelutto et al. [120].

[5] WindFlow library on GitHub: `https://github.com/ParaGroup/WindFlow`

4.5.1 FastFlow features not covered in this thesis

There are important features of the FastFlow library that are not covered in this dissertation. They are: i) the FastFlow memory allocator; ii) the support of GPUs platforms by using either OpenCL or CUDA; iii) the extension of the FastFlow programming model to target distributed-memory systems. In the following, we provide a brief introduction to these features.

FastFlow memory allocator. The FastFlow library provides a custom allocator optimized for the allocation of small objects used in a producer-consumer way. It is based on the idea of *slab* allocator [58]. A slab is a contiguous region of memory split into equal-size chunks plus a header containing information about how many of those chunks are in use. Virtual memory is acquired and released per slab using a general-purpose allocator (by default `libc` malloc/free calls). The allocator is implemented as a C++ class that provides malloc-like and free-like methods.

A set of slabs, for a given object size, are pre-allocated in a *local cache*, so that when a request to allocate memory for an object of that size is received, it can be immediately served by using a free chunk. A request to release an object just produces a new item in the free chunk list without really releasing virtual memory. Only when all the chunks of a slab have been released, the slab memory is returned to the general-purpose allocator. This simple process eliminates the need to search for suitable memory space thus increasing the performance, reduces memory fragmentation and increases memory re-use [58].

The base FastFlow allocator has been implemented with the idea that only one thread can allocate memory (*mem-producer*) and one or more threads can release memory (*mem-consumer*(s)). This is the typical scenario of *Task-Farm* and *pipeline* computations. For implementing these simple scenarios, the FastFlow allocator internally uses lock-free SPSC queues, i.e. the same data structure used in FastFlow to implement the communication channels between building blocks.

The efficiency of the FastFlow allocator in the context of *Data Stream Processing* parallel patterns have been investigated and assessed in Torquati et al. [311].

GPUs targeting. To the best of our knowledge, SkePU [146], SkelCL [304] and Muesli [147] were the first algorithmic skeleton libraries to target the deployment of data-parallel computations to heterogeneous CPU+GPU systems. We took inspiration from those libraries to introduce GPU and multi-GPUs support in the FastFlow library too.

The FastFlow characteristic to wrap arbitrary functions into a sequential building block node facilitates the integration and the use of external accelerators such as GPUs. However, GPUs present additional challenges because of their different features and capabilities, which typically require direct intervention and tuning from the programmer. To facilitate the programmer, both CUDA and OpenCL support have been added to the FastFlow library by leveraging customized sequential building blocks such as the `ff_oclNode` for the OpenCL architecture.

On top of the basic building blocks the Loop-of-Stencil-Reduce (LSR) pattern implemented both in CUDA and OpenCL has been devised as an abstraction for tackling the complexity of implementing iterative data-parallel computations on heterogeneous platforms targeting both CPU cores and GPU accelerators [23, 8]. The LSR pattern can be nested in other stream parallel patterns, such as *farm* and *pipeline*. Aside from the LSR pattern, the more classical data parallel pattern such as *Map* and *Map+Rudce* have also been provided to the FastFlow user. Currently, the kernel code executed by the patterns is wrapped into proper C macros functions.

Distributed FastFlow. FastFlow provides an implementation running on distributed systems [11]. It is currently based on the ZeroMQ communication library, which allows us to build complex asynchronous message-passing networks in an easy way if compared to standard socket programming, yet providing reasonable performance [198].

To support inter-process communication, the FastFlow standard node has been extended with an additional "external channel" (the new channel can be present either in input or in output). The extended node is called *dnode* (distributed node). The basic idea is to allow the edge nodes of the FastFlow network to communicate with the "external world" through a well-defined set of communication collectives. This way, internal communication are implemented through shared-memory FastFlow channels whereas external communications happen through a transport layer and according to useful abstractions for effective programming of hybrid multi-core and distributed platforms.

Name	Communication pattern description
`unicast`	unidirectional point-to-point channel between two *dnodes*
`onDemand`	one-to-many channel; the data is sent to one of the connected peers, the choice is made dynamically on the basis of the actual workload of the connected peers
`scatter`	one-to-many channel; sends different parts of the data to all connected peers
`broadcast`	one-to-many channel; sends the same data to all connected peers
`fromAll`	many-to-one channel; collects different parts of the data coming from all connected peers and then combines them in a single data item (this pattern is also known as *all-gather*)
`fromAny`	many-to-one channel; collects one data item from one of the connected peers non-deterministically

Table 4.2: Available communication patterns between distinct *dnode*s in the distributed FastFlow version.

The communication patterns currently implemented are summarized in Table 4.2. By means of two auxiliary methods provided by the FastFlow *dnode* for data marshalling and unmarshalling, data elements are serialized and then streamed into network channels, de facto extending the internal channel allowing data to flow across *dnodes*. The serialization and de-serialization methods slightly increase the coding complexity but they support the development of complex distributed applications. There exists also a proof-of-concept implementation of a minimal message passing layer implemented on top of InfiniBand RDMA features which can be used in alternative to ZeroMQ [292]. The proposed RDMA-based communication channel implementation achieves comparable performance with optimised MPI/InfiniBand implementations.

In this thesis, we do not cover the programming model and the implementation features of the distributed version of FastFlow. However, we want to remark here that the FastFlow model is suitable to be extended to target distributed systems to provide the user with a unified model for shared-memory multi-cores and distributed memory systems. In this respect, the FastFlow model and its distributed prototype, has inspired the interesting research work done at University of Turin about the GAM model, where executors (i.e. the equivalent of FastFlow *dnodes*) synchronize with each other by exchanging messages that are global memory references enriched with access attributes [143].

4.6 Summary

In this chapter, we introduced the new FastFlow parallel library (version 3) and its programming model. The FastFlow library is the result of a research effort started in 2010 as a joint work between people from the Computer Science Department of University of Pisa and Turin. Over the years it has been improved and extended thanks also to its employment as RTS into three EU-funded research projects (ParaPhrase, RePaRa and RePhrase).

The new version of the library introduced in this thesis extends the previous version mainly in two directions: 1) the redesign of the lower software layer of the library introducing also new parallel building blocks that strengthened the flexibility of the entire library and opens new opportunities to build more scalable parallel solutions; 2) the introduction of a new concurrency graph transformation component that enables the refactoring of the concurrency graph describing the FastFlow parallel application with the objective to introduce optimizations.

We believe, supported by the many research works and applications developed in the last years, that the FastFlow building block software layer is mature and stable. We do not foresee the addition of new building blocks into the current set. The new concurrency graph transformation layer added into the library will allow us to systematically design and implement several well-known and also new static optimizations that so far have been implemented mainly manually by the FastFlow programmer. This software layer is particularly important to enhance the *Performance portabil-*

ity of the library for future multi-core systems with a high number of heterogeneous cores. In our opinion, the new features we have introduced in the new FastFlow library will foster new research directions in the context of structured parallel programming optimizations and DSLs development.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Building Blocks Definition

5.1 Introduction

Several different approaches have been proposed to tackle the abstraction difficulties in developing parallel applications on modern heterogeneous multi-core architectures. One of the most popular approaches is the one based on sequential code annotation where programmers use compiler directives to annotate regions of code that can be executed in parallel either on the available CPUs or, for example, on a GPU device. Then, the compiler automatically and transparently uses these directives to generate efficient run-time code targeting code execution on the selected CPU or GPU. Currently, the reference programming models using code annotations are OpenMP [53] and OpenACC [263].

If on the one hand such parallelization approaches increase the level of abstraction relieving the programmer to deal with resources allocation, data transfers, and explicit synchronizations, on the other hand, they make it difficult to fine-tune and customize the parallel application for specific needs. In fact, since the parallel structure of the application is not explicitly exposed to the programmers, this makes it difficult to use such libraries as RTS of more abstract and higher level framework.

Conversely, the structured parallel programming approach (whose foundations are based on *algorithmic skeletons* [94]) makes explicit to the programmer the parallel structure of the application promoting the introduction of parallelism since the first phases of the application development process hiding all parallelism exploitation details. Following the principles of this parallel programming model, a parallel application is conceived by selecting and assembling a small set of distinct components to avoid introducing any unneeded dependencies for both data and control flows. *Functional* and *performance portability* (see Section 2.4.1) across different target platforms are then obtained through abstraction of the underlying architectures. For each parallel pattern, efficient and parametric implementations of both communications/synchronizations and computation are available (i.e. algorithmic skeletons). The QoS predictability of these parallel schemes can then be used to evaluate their profitability, producing the best configuration and enabling adaptivity support and autonomic solutions.

Accordingly, we envision the use of *parallel building blocks* as fundamental components to build ready-to-use parallel patterns. The objective is to promote the so-called LEGO-style approach to parallel programming. However, building blocks alone, do not capture the whole parallel structure of a parallel application. Rather, they should be used in proper compositions to model the parallel structure of the given application. The level of abstraction offered by the building blocks is lower than the one provided by high-level parallel patterns. Nevertheless, they still promote *performance portability* thanks to a precise functional and parallel semantics. In addition, building blocks are intended to be used by RTS programmers to provide high-level abstractions (i.e. new parallel patterns) to the application programmers. This idea can be considered at the core of the structured parallel programming methodology. It promotes the skeleton/pattern approach not only at the application level to ease the application programmer's job, but also in the design and implementation of the run-time support by providing powerful parallel abstractions capable of simplifying RTS developer's job too.

In Section 5.2 we introduce the sequential and parallel building blocks provided by the FastFlow library. Section 5.3 describes the rules for combining and nesting the building blocks. Section 5.4 shows how a possible implementation of the BSP parallel programming model can be described by using the FastFlow building block.

5.2 FastFlow building blocks

To describe the building blocks, we propose an analogy with the famous LEGO bricks. LEGO bricks are simple components with different shapes (square, rectangular, circular, etc.) each with a different number of studs[1]. Studs fit into grooves found at the bottom of every brick to create new shapes. LEGO pieces can be assembled and connected in many different ways to construct complex objects including castles, robots, and spaceships. Everything built by using LEGO bricks can then be taken apart and reassembled differently to modify or improve the created objects.

Following this analogy, final LEGO objects are the parallel applications while the basic LEGO bricks are the FastFlow building blocks. Just as LEGO bricks are used to create new shapes, in the same way, the FastFlow building blocks are used to implement parts or the whole parallel application. The programmer (either the application programmer or the RTS developer) selects the most suitable building blocks and assemble them in a smart and efficient way to solve the problem at hand. In our vision, FastFlow building blocks are those concurrent components that are the fundamental elements of any structured parallel applications. In Chapter 9 we will see that carefully designed compositions of building blocks allows us to obtain powerful high-level parallel patterns. In Chapter 8 we will show how the building blocks composability feature can be effectively employed to transform the concurrency graph of a pattern-based application by preserving its functional semantics and improving its performance.

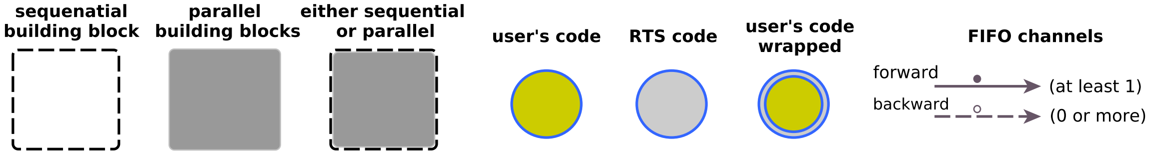


Figure 5-1: Symbols used to describe FastFlow building blocks.

To represent the FastFlow building blocks, we will use the diagrams sketched in Figure 5-1. The basic abstraction of the building block set is the *node*. It encapsulates either user's code (i.e. business logic code) or RTS code. User's code can also

[1] A stud is the little bumps on the top of almost every LEGO brick

be wrapped by a FastFlow node executing RTS code (the diagram with two nested cicles in Figure 5-1). In this way, input and output data can be manipulated and filtered before and after the execution of the business logic code. This feature is particularly useful to interface already written code as well as to apply some automatic transformations (cf. Chapter 8). A sequential *node* defines the unit of execution in the FastFlow framework. The diagrams used to represent valid sequential nodes are shown in Figure 5-2. User or RTS code can be embedded in a *sequential* or a *parallel* building block that defines its concurrent behavior over a single or multiple input data element(s) (i.e. streams) carried by *channels* which connect *nodes*.

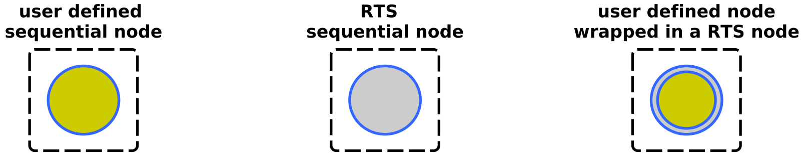


Figure 5-2: Valid sequential nodes.

The set of FastFlow building blocks are shown in Figure 5-3. We will refer to them also as *FF-bb*. Building blocks are either *sequential* or *parallel*. Sequential building blocks are nodes with one or more input or output channels. Parallel building blocks are concurrent components made out of a proper assembly of nodes and point-to-point FIFO channels. We will show that *FF-bb* can be composed and nested using a LEGO-style programming model to build more complex parallel components with data-flow semantics. Note that there is always at least one forward channel connecting two building blocks (or two nodes of a building block) whereas there could be zero or more backward channels (this is denoted using a small empty circle close to the dashed arrow representing a backward channel).

Sequential nodes. Sequential FastFlow nodes are concurrent entities that can be classified according to their number of input and output channels. We define *input cardinality* and *output cardinality* of a building block the number of input and output channels, respectively. It is possible to distinguish four kinds of sequential nodes:

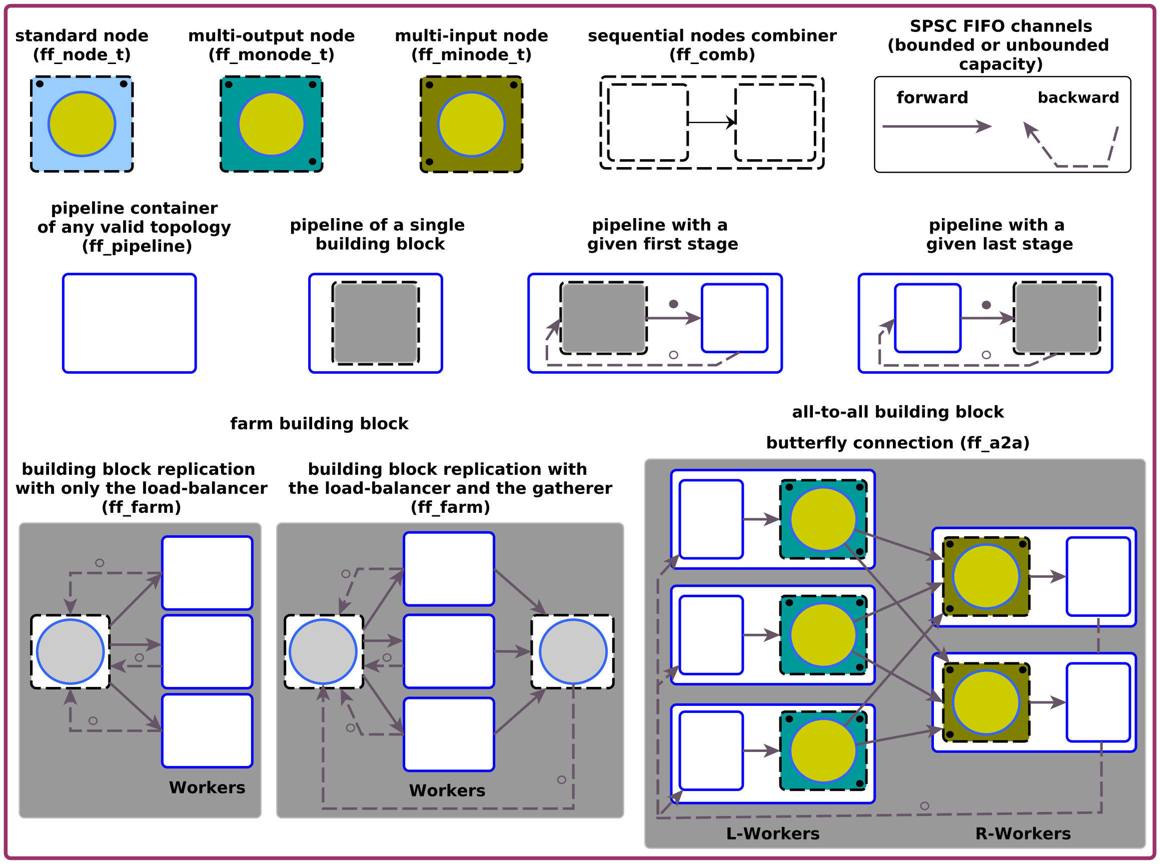
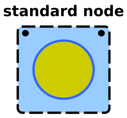
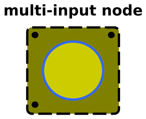


Figure 5-3: The FastFlow building blocks.

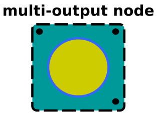
1. the *standard node* (or simply *node*) with input and output cardinalities equal to one;



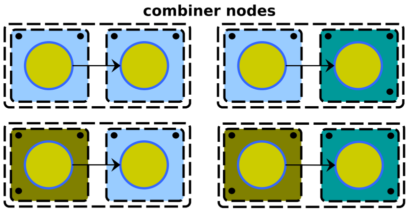
2. the *multi-input node* with the input cardinality *not statically defined* and the output cardinality equal to one;



3. the *multi-output node* with the input cardinality equal to one and the output cardinality *not statically defined*;



4. the *sequential node combiner*, which allows us to combine two sequential nodes into a single node.



A node with a cardinality "*not statically defined*" (either in input or in output) is a node whose input or output cardinality will be defined when the node is connected to another node. As we shall see, this distinctive property of *multi-input* and *multi-output* nodes enables great flexibility in connecting building blocks. As an example, a *multi-output* node can be connected to whatever building block with any input cardinality. When the network of nodes describing the application has been completely built, the *multi-output* node will inherit the input cardinality of the building block to which it is connected to. A *standard node* might have no input channel or no output channel. In the first case, it acts as a *source* node (i.e. a node generating data items). In the second case, the node acts as a *sink* node (i.e. a node absorbing data items).

A *combiner node* is a sequential building block aiming at promoting *code reuse* through fusion of already existing nodes. It can also be used to reduce the concurrent resources of the data-flow network by decoupling the node abstraction from its concrete implementation. Finally, it can be used to "adjust" the input or output cardinality of an existing node to connect it to a *multi-input* or *multi-output* parallel building block.

As we shall see in Chapter 7, a FastFlow node is implemented as a C++ object. Therefore, by leveraging C++ object inheritance and methods overloading features, and thanks to the possibility of redefining some of the methods for controlling the output message routing, it is straightforward to build custom sequential nodes.

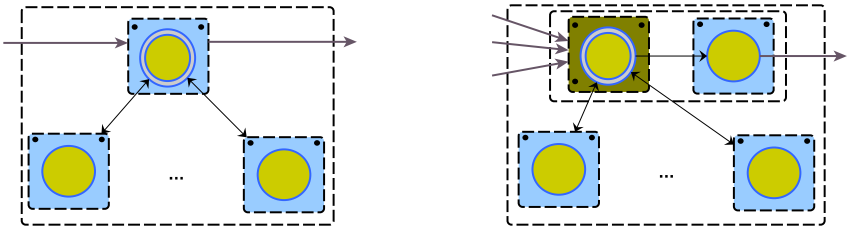


Figure 5-4: Two simple examples of custom nodes.

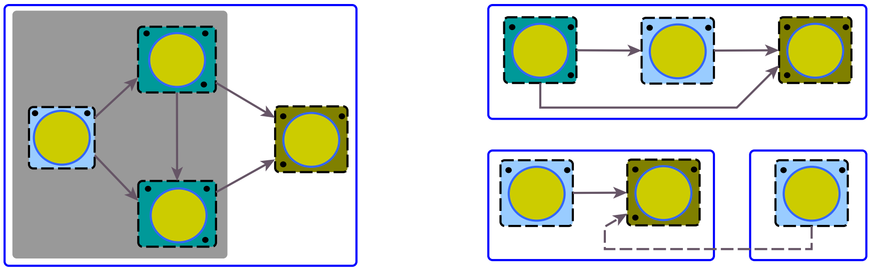
As an example, Figure 5-4 shows two simple cases of "selector node", i.e. a node that encapsulates a number of other nodes and that dynamically (or even statically) selects toward which node the data elements received in input have to be forwarded. The results produced by the selected node are then diverted to the proper output channel(s). Such flexibility, allows the RTS programmer to build his/her own parallel abstractions readily. This is a leading feature of the FastFlow building block which promotes *programmability* even at RTS level. As a side note, a specific implementation of the "selector node" has been provided in the RePaRa project run-time based on the FastFlow library (see Section 3.3.5). In that case, a "selector node" was used for targeting multiple devices (i.e. GPUs, FPGAs, and DSPs) for a given kernel function [237].

Channels. Communication channels are used for connecting two building blocks as well as the internal nodes used for implementing a parallel building block. We distinguish between *forward* and *backward* channels (also called *feedback* channels) on the basis of the respective position of the sender node and receiver node with respect to the conventional flow of data in a pipeline composition of nodes. All communication channels are implemented employing Single-Producer Single-Consumer (SPSC) FIFO-ordered queues (see Chapter 6 for the implementation details). FastFlow channels do not carry plain data, but references to heap allocated data. Communication channels are also used as synchronization mechanisms between two distinct FastFlow nodes accessing a shared data structure. The semantics of sending references over a channel is that of transferring the ownership of the value pointed from the sender node to the receiver node. Therefore, the receiver is expected to have exclusive access to the data value. The capacity of a channel connecting two nodes can be either *bounded* to a fixed value or *unbounded.* Feedback channels always have an *unbounded capacity* to avoid deadlock. The implementation of communication channels and the concurrency control policies used to coordinate channel accesses are described in Chapter 6.

Parallel building blocks. The parallel building blocks are the *pipeline*, the *all-to-all* and the *farm.* The latter has two different topologies (see Figure 5-3). The *pipeline* building block is used both as a container of building blocks (including the pipeline itself) and as application topology builder. At run-time, the *pipeline* building block models data-flow computations working on streams. The *farm* and *all-to-all* building blocks model functional replication with and without a centralized coordination, respectively. The *farm* defines a single set of Workers with a centralized coordinator called Emitter, whereas the *all-to-all* defines two distinct sets of *Workers* connected together according to the *shuffle* communication pattern (see also Section 2.5.3). We will describe the parallel semantics and the implementation of each parallel building block in Chapter 7. Figure 5-3 shows that farm's Workers, as well as the L-Workers and R-Workers of an *all-to-all* building block, can be any valid *pipeline* composition of building blocks. For the *all-to-all*, the only constraint is that the left-most node of the L-Workers component must be a multi-output node and the right-most node of the R-Workers component must be a multi-input node. Concerning the *farm*'s Emitter, it can host any sequential node. However, as we shall see in Chapter 7, the Emitter is a specialization of a multi-input sequential node.

Connecting building blocks. Sequential and parallel building blocks can be connected (through one or more SPSC FIFO channels) by adding them to a *pipeline* building block in the desired order. Not all possible compositions of *FF-bb* are allowed. In Section 5.3 we will define the rules that allow building only well-defined pipelines.

Despite the name, the pipeline building block permits to assemble either concurrent linear chains of nodes (i.e. standard data-flow pipelines) as well as more complex (unstructured) topologies (e.g., multi pipelines). However, not all possible directed graphs can be constructed by using the *FF-bb*. For example, the following graphs of nodes are not valid compositions of FastFlow building blocks:



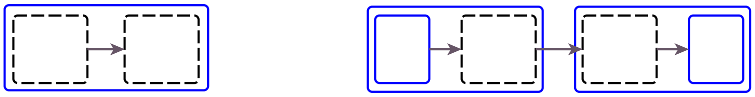
Notwithstanding, the expert programmer can always use the FastFlow channels to connect independent building block "manually" without the coordination of the *pipeline* building block. We do believe that this operation is typically not needed and, although supported, it has to be considered as a last resort and an "escape strategy".

Building blocks added to a *pipeline* will be executed according to the data-flow parallel semantics. Besides, as long as the pipeline is not started, it acts also as a container of building blocks whose input and output interface are the ones of the first and last building blocks added to the pipeline itself. Therefore, according to the implicit grammar described in Figure 5-3, a pipeline can be used as a base component to build complex topologies made of multiple nesting of pipelines with several different building blocks inside. It is worth pointing out that a pipeline may have different input and output cardinalities depending on its first and last stage.

5.3 Composition rules

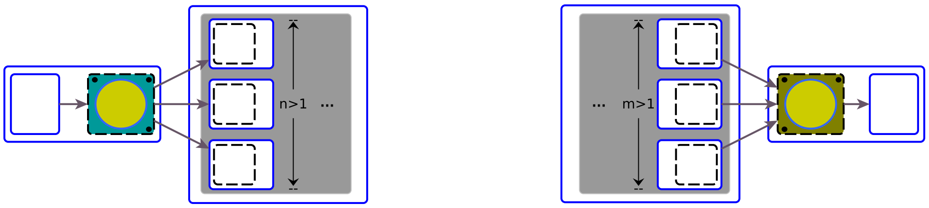
The FastFlow building blocks can be composed and nested to build networks of nodes which are executed according to the data-flow model. The rules for connecting building blocks and to generate only valid network topologies, are as follows:

1. Two sequential building blocks can be connected in pipeline regardless of their input/output cardinality:



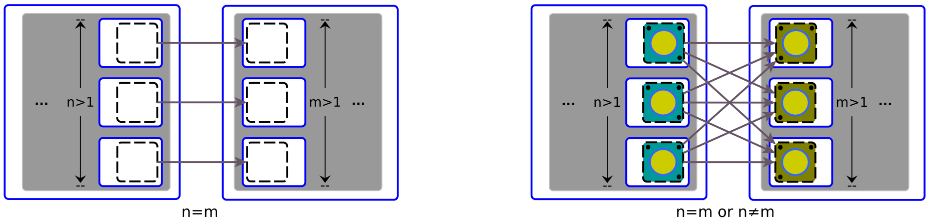
   A particular case is the conposition or *multi-output* and *multi-input* nodes. In this case we assume a cardinality of one for the *multi-input* node.

2. A parallel building block can be connected to a sequential building block through a *multi-output* and a *multi-input* nodes:



3. Two parallel building blocks can be connected either if they have the same

number of nodes regardless of their input/output cardinality or through *multi-input multi-output* nodes if they have a different number of nodes:



If the edge nodes of the two parallel building blocks are *multi-output* and *multi-input* respectively (the right-hand side schema above), then, when they are connected, any edge nodes of the first building block will be connected to any other nodes of the second building block according to the *shuffle* communication pattern.

The input and output channel cardinality of a single building block or of a composition of building blocks can be computed considering the rules showed in Figure 5-5. We used the symbol "*" to denote a cardinality that is "not statically defined".

It is worth observing that a given building block may have different cardinality on the basis of the number of edge-nodes composing it and on the basis of the sequential node used for implementing edge-nodes. For example, a farm without Collector may have an output cardinality equal to its number of Workers only if its Workers are standard nodes. However, if the Workers are multi-output nodes, the farm has a cardinality equal to the product of its number of Workers by the output channel cardinality of each Worker. Furthermore, the farm building block has an input cardinality that is "not statically defined". This means that the farm behaves as a *multi-input* node. For example, the farm can be used to receive output data from another farm without the Collector or from an *all-to-all* building block with multi-output R-Workers.

Given the composition rules described above, it is easy to see that the topologies described in Figure 5-6 are not allowed within the FastFlow library. In general, it is considered *not-valid* any pipeline composition where *both* the following conditions hold:

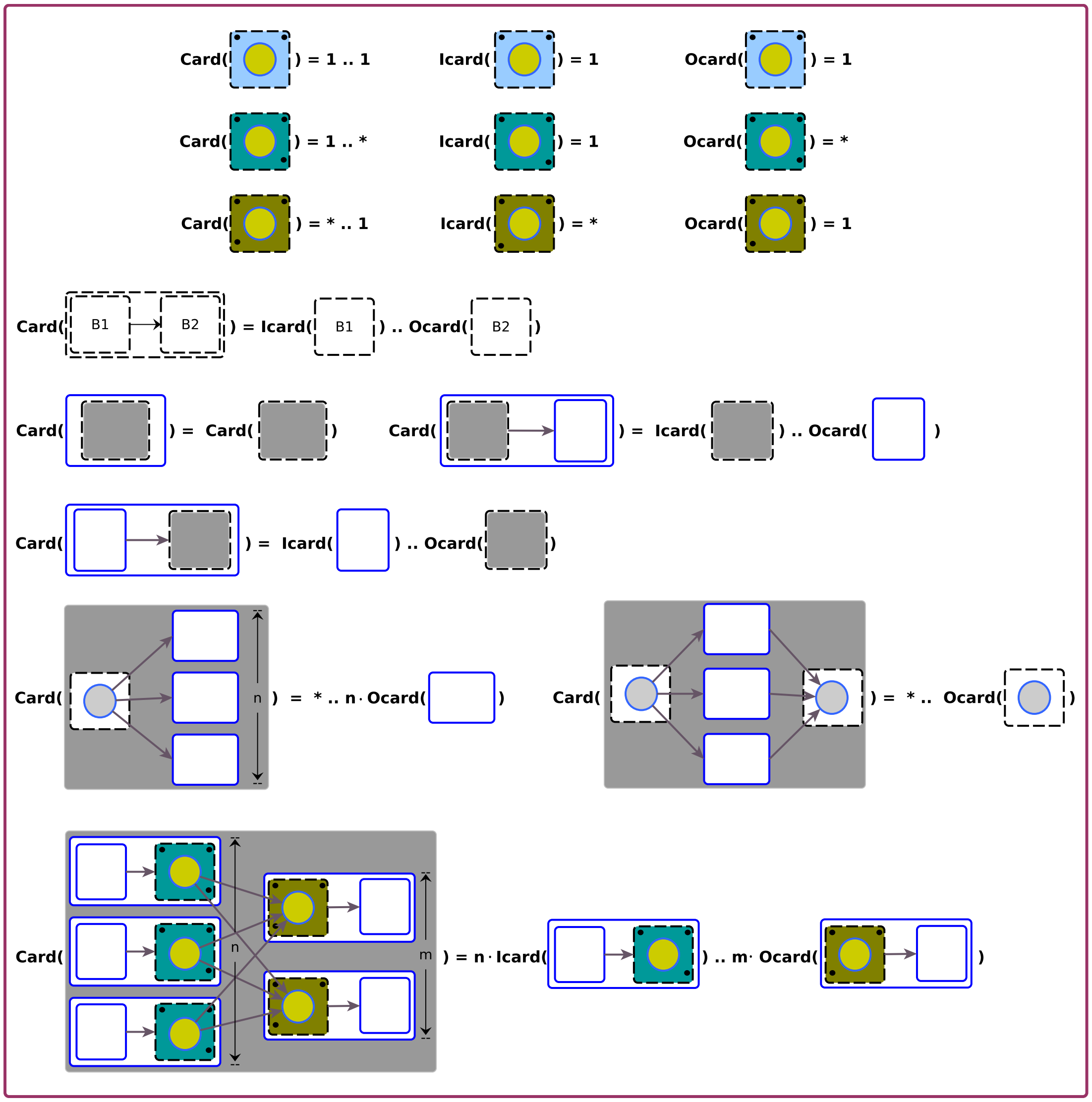

Card( ) = 1 .. 1
Icard( ) = 1
Ocard( ) = 1
Card( ) = 1 .. *
Icard( ) = 1
Ocard( ) = *
Card( ) = * .. 1
Icard( ) = *
Ocard( ) = 1
Card(B1 B2) = Icard(B1) .. Ocard(B2)
Card( ) = Card( )
Card( ) = Icard( ) .. Ocard( )
Card( ) = Icard( ) .. Ocard( )
Card(n) = * .. n · Ocard( )
Card( ) = * .. Ocard( )
Card(n m) = n · Icard( ) .. m · Ocard( )

Figure 5-5: Cardinality of each building block. *Icard*/*Ocard* denote the Input/Output cardinality, respectively. The function *Card* returns both cardinalities using the notation: *Card(BB) = Icard(BB) .. Ocard(BB)*.

(i) the edge-nodes of the two building blocks being connecting have a different cardinality; **and**

(ii) the edge-nodes of the left-hand building block are not multi-output and the edge nodes of the right-hand building block are not multi-input.

Essentially, having a mismatching cardinality between the output of a building block and the input of another building block is a sufficient condition for not connecting the two only if *multi-input* and *multi-output* nodes are not used at the edge

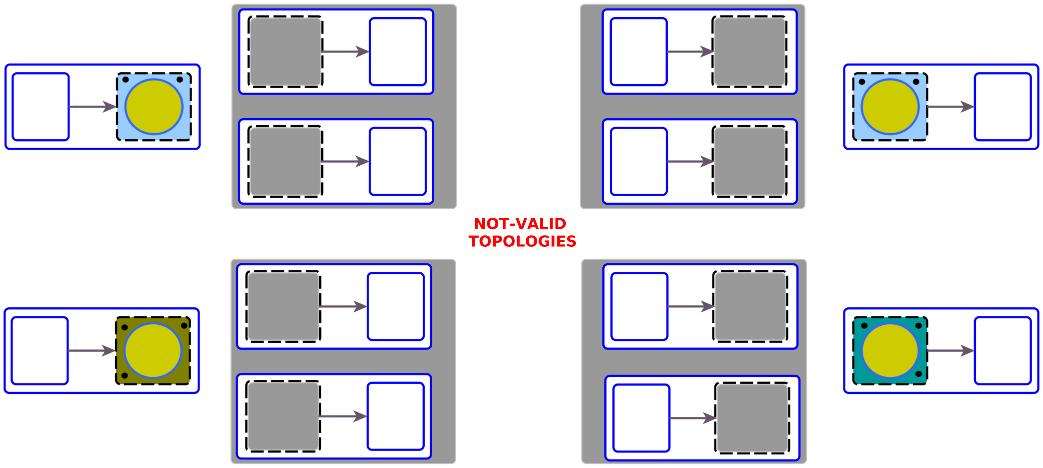


Figure 5-6: Building block compositions that are not allowed.

nodes of both building blocks. Such a strong connectivity policy is due to multi-output and multi-input nodes which connect themselves greedily to other nodes. Moreover, the two previous conditions imply that two building blocks can be always connected provided that their edges nodes are multi-output and multi-input, respectively.

Some examples of valid pipeline compositions of different building blocks are shown in Figure 5-7. The top most composition (composition a) in the figure) is a three-stage pipeline of five sequential nodes. The middle stage is implemented as a *sequential nodes combiner* building block of three standard nodes. Composition b) is a pipeline of three building blocks with a feedback channel between the last and first stage. The first stage is a sequential node; the second stage is a *farm* without the Collector and whose nodes are *pipeline*s of two sequential FastFlow nodes. Finally, the last stage is a *farm* with the Collector whose nodes are multi-output nodes having a forward channel toward the Collector and a feedback channel toward the Emitter. When the pipeline composition in Figure 5-7(b) is executed, it will have six pipeline stages and a concurrency degree of eleven distinct nodes (considering two Workers in the first *farm* and three Workers in the second *farm*). Composition c) in Figure 5-7 is still a three-stage pipeline network whose first stage is a *farm* of multi-output nodes (with feedback channel toward the Emitter), the second stage is an *all-to-all* building block. The last stage of the pipeline is a sequential *multi-input* node. In this case we can see how the *multi-output* Workers of the first *farm* are connected to all L-Workers

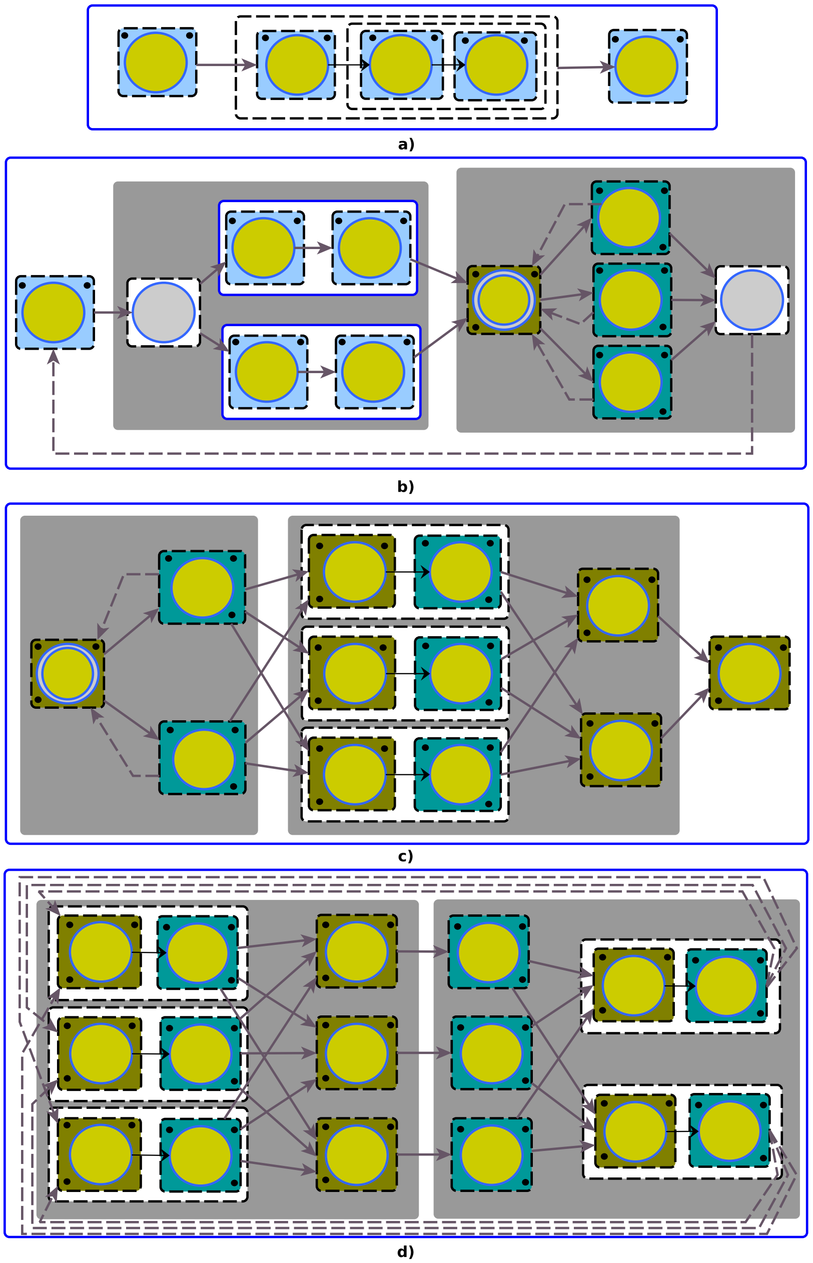


Figure 5-7: Some examples of building block compositions.

of the *all-to-all* even if they have different numbers of Workers. Moreover, since the L-Workers of an *all-to-all* building block must be multi-output nodes, to realize a valid connection between the two building blocks, a *sequential nodes combiner* can be used for implementing a sequential node that is both *multi-input* and *multi-output* at the same time. Finally, composition d) shows a pipeline of two *all-to-all* whose L-Workers are *multi-input* nodes while R-Workers are *multi-output* nodes. There is a feedback channel connecting all R-Workers of the second *all-to-all* with the L-Workers of the first *all-to-all*. In addition, since the number of R-Workers of the first *all-to-all* is equal to the number of L-Workers of the second *all-to-all* there is a direct single connections between the workers of the two *all-to-all* (to implement a shuffle patterns between the two *all-to-all* it is sufficient to combine a *multi-output* node and a *multi-input* node to the workers at the two respective edges).

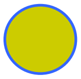
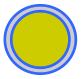
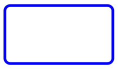
The examples presented in Figure 5-7 show the expressive power of the *FF-bb* for constructing sophisticated data-flow topologies of concurrent nodes. The proposed schemas can be simply obtained by adding building blocks to a pipeline container paying attention to the small set of composition rules discussed in this section.

We remark that the high degree of freedom offered by the FastFlow building block software layer enables both modular programming and user's code reuse. Besides, it allows us to expand the design space exploration phase by quickly moving from one implementation to a different one without the need to change the business logic code of already tested parallel components thus limiting the programmer intervention only at the edge nodes of the topology. Finally, even though applications built by using high-level parallel patterns typically do not present complex process network topologies, the process topologies reported in Figure 5-7, which present more elaborate connections, can be the result of a parallel program optimization phase performed according to some heuristics and through multiple concurrency graph transformations (we will discuss some graph transformations in Chapter 8). On the other hand, complex typologies may be the result produced by the RTS programmer that is designing a new domain-specific pattern.

5.3.1 RISC-pb^{2}l equivalence

The set of FastFlow building blocks presented in the previous sections has been heavily inspired by the RISC-pb^{2}l building blocks recently proposed in our previous research work [118, 7].

RISC-pb^{2}l is a description language conceived to reason about structured parallel programming. One of its primary objectives is that of allowing the skeletons/patterns

RISC-pb^{2}l	FastFlow building blocks
Seq Wrapper	
Par Wrapper	
Parallel and MISD	either farm or all-to-all building blocks
Pipe	

Reduce
(L-level K-ary tree)
L,R ⩾ 1
	if L=1, a single multi-input node
	if L>1, a L-1 nesting of all-to-all with K L-Workers and 1 R-Workers

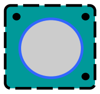
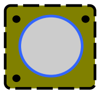
Spread (L-level K-ary tree) L,R ⩾ 1	if L=1, a single multi-output node if L>1, a L-1 nesting of farms each with K Workers
1-to-N	
N-to-1	
feedback	feedback channel applied to pipeline, farm, all-to-all and their compotions

Figure 5-8: Correspondence between the RISC-pb^{2}l and the FastFlow building blocks.

methodology to percolate to a lower level of abstractions than that of the applications. The intent is that of studying code refactoring and to introduce static optimizations into the developed application. A description of the RISC-pb^{2}l grammar may be found in Section 2.5.4 (Figure 2-16). All RISC-pb^{2}l building blocks can be implemented by using the FastFlow building blocks. Their mutual equivalence is shown in Figure 5-8. The RISC-pb^{2}l *Parallel* building block can be implemented with either a FastFlow farm or with an *all-to-all*. The choice of which one to use depends on the other building blocks used to define the application graph. For example, a single *all-to-all* with the same number of L- and R-Workers can be used to implement a pipeline of two *Parallel* building blocks. The *Spread* and *Reduce* functionals can be implemented by a proper nesting of the *farm* and of the *all-to-all* building blocks as sketched in

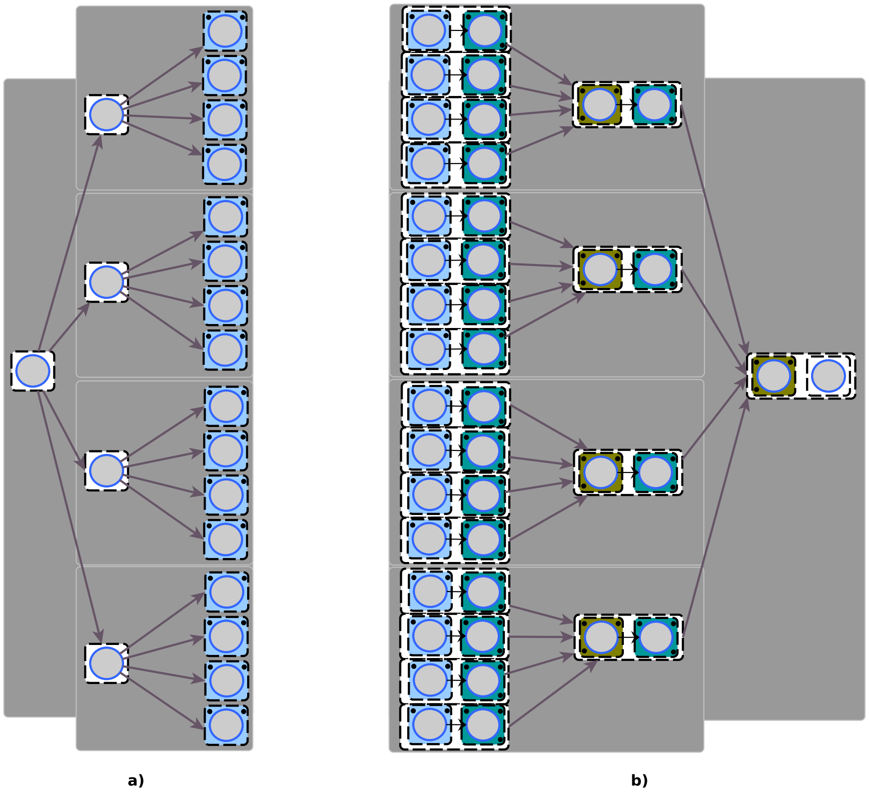


Figure 5-9: FastFlow building blocks implementation of the 3-level 4-ary tree *Spread* (a) and *Reduce* (b) functionals of RISC-pb^{2}l.

Figure 5-9.

The main differences between *FF-bb* and RISC-pb^{2}l can be summarized as follows:

- *FF-bb* has a primitive building block (the *all-to-all* one) that is not present in RISC-pb^{2}l whose peculiarity is to have an input cardinality of n and an output cardinality of m, where possibly $n \neq m$ and $n > 1, m > 1$. The *all-to-all* building block permits to build pipeline composition with different input and output cardinalities. Instead, RISC-pb^{2}l accepts as valid composition only the following ones: $\Delta^{1n} \bullet \Delta^{n} \bullet \Delta^{n1}$ and $\Delta^{1n} \bullet \Delta^{n1}$.

- *FF-bb* provides more flexibility when connecting building blocks because of the "*not statically defined*" cardinality of *multi-input* and *multi-output* nodes. For example the *FF-bb* program described in Figure 5-10 cannot be written in RISC-pb^{2}l because it is not possible to connect a *N-to-1* building block with both a *Parallel* and a *Sequential wrapper*.

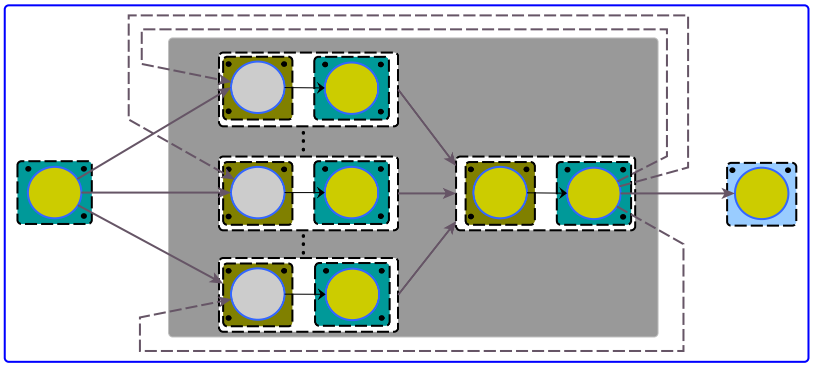


Figure 5-10: FastFlow building block program that cannot be expressed using the RISC-pb^{2}l building blocks.

A valid RISC-pb^{2}l program emulating the *FF-bb* topology described in Figure 5-10 could be:

$$\overleftarrow{(\triangleleft_{Pol} \bullet [|\ \Delta\ |]_n \bullet \triangleright_{Pol} \bullet ((code)))}_{cond}$$

- *FF-bb* offers a clearer distinction between sequential and pipeline composition than RISC-pb2l. In *FF-bb*, the first is used to reduce the number of active concurrent entities and to increase computation granularity, the latter to express pipeline concurrency.

5.4 BSP model implementation

As a more complex and relevant example, we show how the *Bulk Synchronous Parallel* (BSP) model can be implemented by using the FastFlow building blocks. The BSP general model of computation is described in Section 2.4.3.

At a high level, a BSP computation can be described as a sequence of *supersteps* each one having a parallel computation phase and a communication phase. All BSP processors run the same program according to the SPMD paradigm, and each processor can independently send/receive data to/from other processors. At the end of each superstep, a bulk synchronization (i.e. a barrier) is performed ensuring that all communications have been concluded.

From the FastFlow building block viewpoint, each superstep can be seen as a two-

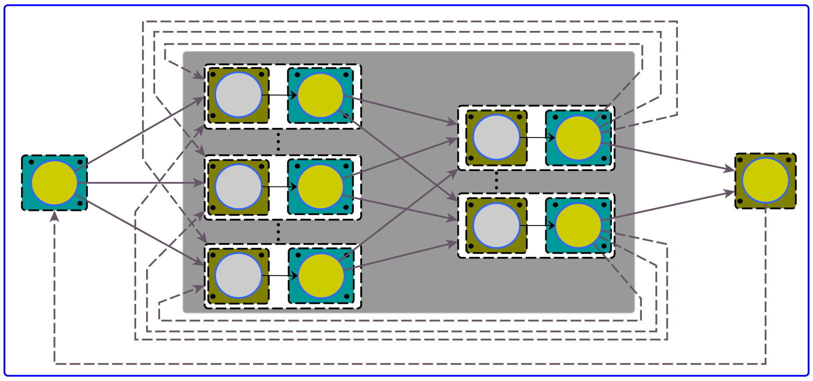


Figure 5-11: Building block implementation schema of the BSP model.

stage pipeline in which the first stage of the pipeline executes the computation phase while the second stage implements the communication and barrier features of each superstep. The single superstep may then be modeled using the *all-to-all* FastFlow building block with feedback channels between R-Workers and L-Workers. The feedback channel allows implementing the sequence of supersteps needed to compute the final result. The L-Workers are the processors of the BSP model. They are responsible for executing the computation phase and sending the partial results computed to the R-Workers that act as “communication processors”. L-Workers produce a list of pairs `<value,index>`, where the *index* denotes the id of the p BSP processor to which the data `value` has to be directed for computing the next superstep. The L-Workers send the result produced to a subset of R-Workers selected, for example, by using a round-robin or hashing policy, and send just a synchronization message to all other R-Workers not part of the receivers set. The number of R-Workers to use and the size of the receivers set are configuration parameters that depend on the target machine. However, in most cases, it is reasonable to assume that the number of R-Workers is a small subset of the number of L-Workers and that the cardinality of the receivers set is a small number (e.g., 1 or 2). The R-Workers are used to implement the barrier and also to combine partial results received by the L-Workers to minimize the number of messages that have to be sent when redistributing the collected data back to the L-Workers for computing the next superstep (if any). In this way, it is possible to combine both data distribution and synchronization to implement the required BSP barrier.

It is worth remarking that in the *FF-bb* implementation on multi-core platform data is not moved. Instead, references to data are passed through communication channels (see Chapter 6 for an in-depth discussion about FastFlow channels). The *FF-bb* implementation of the BSP model is sketched in Figure 5-11.

The first stage of the pipeline is responsible for partitioning the initial dataset to the L-Workers. It also starts program termination on the basis of the feedback received from the last stage of the pipeline. Actually, managing program termination when the number of supersteps is not known in advance, requires extra synchronizations that are implemented by using the third stage of the pipeline. Specifically, when there are no more data to redistribute by all R-Workers, the third stage of the pipeline will notify the first stage that thus will notify all L-Workers to terminate.

5.5 Summary

The definition of suitable and performant parallel abstractions with their associated functional and parallel semantics requires efficient parallel RTSs capable of delivering adequate performance, possibly close to hand-tuned solutions. One of the challenges is to define efficient and flexible primitive components and a clear methodology for designing new parallel abstractions.

In this chapter, we defined a small set of building blocks with their functional and parallel semantics. Building blocks are primitive sequential and parallel components that can be combined and nested in many different ways according to a small number of composition rules. We described the semantics of each building block together with how it can be composed to other building blocks according to the rules defined. We also identified which process network topologies cannot be constructed with the set of building blocks proposed. Finally, we showed the equivalence of the FastFlow building blocks with the RISC-pb2l set of parallel components [118, 7], which can model some of the most powerful parallel patterns and parallel programming abstractions (e.g., Google's Map-Reduce [134]). The building blocks set proposed in this chapter is a superset of the RISC-pb2l set. The new set includes the new *all-to-all* parallel building block modeling the *shuffle* communication pattern and powerful composition rules. In the remaining Chapters, we will see how the proposed set of primitive components promotes the structured parallel programming methodology at the RTS level of the software stack encouraging a LEGO-style approach for the design and implementation of parallel patterns.

Chapter 6

Communication Channels

6.1 Introduction

In this chapter we describe the FastFlow communication channels that are, according to the analogy between FastFlow building blocks and LEGO bricks we made in Chapter 5, the equivalent of studs and grooves present in almost all LEGO bricks.

In the context of data-flow parallel programming model, communication channels used to connect concurrent entities (*nodes* according to the FastFlow terminology) play a crucial role both for performance and program correctness reasons.

A communication channel connecting two FastFlow nodes is implemented by using a *1-to-1* First-In First-Out (FIFO) queue. With the notation *1-to-1* we intend that, at any point in time, there exist only a Single-Producer (SP) and a Single-Consumer (SC) performing operations on the same queue concurrently. These kinds of queues are also referred to as *SPSC queues*. In principle, a given concurrent entity can perform both roles provided that the SPSC semantics is respected. The producer entity must always invoke a `push` (or enqueue) method whereas the consumer entity must always invoke a `pop` (or dequeue) method.

FIFO queues with their different concurrency levels (i.e. Multi-Producer Multi-Consumer, Multi-Producer Single-Consumer, Single-Producer Multi-Consumer, and Single-Producer Single-Consumer) are an important abstract data structure lying at the heart of most operating systems and application software. Their efficient design has been widely investigated in the scientific works [249, 165, 255, 327, 33].

SPSC queues are particularly interesting because they can be implemented in a very efficient way on multi/many-core platforms without the need for atomic operations [165]. The concurrency control policy used to regulate concurrent accesses to the channel and to handle idle states may have a significant impact on both performance and power consumption of the system. The FastFlow channel has been implemented to support both *blocking* and *non-blocking* concurrency control policies and provides the necessary hooks to support the dynamic switching between the two concurrency modes.

This chapter proceeds as follows. In Section 6.2 we discuss concurrency control policies and their impact on both performance and power consumption, in Section 6.3 we discuss the FastFlow implementation of both bounded and unbounded SPSC FIFO queues used for implementing communication channels. Finally, in Section 6.4, we present how both *blocking* and *non-blocking* concurrency control policies are employed in the FastFlow framework for improving performance and power efficiency.

For the evaluation reported in this and other chapters, we used three different multi-core platforms. They are described in Table 6.1

6.2 Concurrency Control

The standard approach to synchronize the execution of concurrent threads accessing shared data structures consists in protecting the access by using *mutex*-based mechanisms. If the thread that currently holds the mutex is delayed, all the other threads attempting to access the data structure are delayed too. Acquiring the mutex typically implies *passive waiting*, i.e. the suspension of all the threads waiting for the mutex acquisition. The suspended threads are moved in a waiting queue and their core/hardware contexts are released to the OS. However, suspension and restart mechanisms may reduce thread reactivity and therefore application performance due to many factors such as the waiting time in the ready queue, context switch overhead, compulsory cache misses or thread migration [153]. A concurrent algorithm that may

Name	Description	Configuration
Xeon	Dual-socket NUMA machine with two Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40GHz featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared with the cores on the same socket. The machine has 64GB of DDR3 RAM.	Linux 3.14.49 x86_64 shipped with CentOS 7.1. Available compiler `gcc` version `6.4.0`.
KNL	Machine with the Intel Xeon Phi model 7210 (codename Knights Landing, KNL). The KNL is equipped with 32 tiles (each with two cores) working at 1.3 GHz, interconnected by an on-chip mesh network. Each core (4-way Hyper-Threading) has 32 KB L1d private cache and a L2 cache of 1 MB shared with the sibling core on the same tile. The machine is configured with 96 GB of DDR4 RAM with 16 GB of high-speed on-package MCDRAM configured in cache mode.	Linux 3.10.0 x86_64 shipped with Centos 7.2. Available compiler `gcc` version `7.3.0`.
Power8	Dual-socket IBM server 8247-42L with two Power8 processors each with ten cores organized in two CMPs of 5 cores working at 3.69GHz. Each core (8-way SMT) has private L1d and L2 caches of 64 KB and 512 KB, and a shared on-chip L3 cache of 8 MB per core. The total number of cores is 20 physical and 80 logical cores. The machine has 64 GB of RAM.	Linux 4.4.0-47 ppc64 shipped with Ubuntu 16.04. Available compiler `gcc` version `5.4.0`.

Table 6.1: Platforms used for the evaluation of the FastFlow library.

force the calling thread to be blocked waiting for the completion of a given operation on the concurrent data structure is defined as *blocking*.

Concurrent queues can be implemented in an efficient and scalable way by using *non-blocking* algorithms. Although many definitions exist and have been utilized in the literature, here we consider a concurrent algorithm as *non-blocking* when thread suspension cannot be caused by synchronizations related to the data structure usage (of course a thread can still be de-scheduled by a time-sharing scheduler or due to preemption). Therefore, a solution that replaces passive waiting phases with a busy-waiting *spin-loop* (e.g., by replacing mutexes with spin-locks) are accepted as *non-blocking* according to this definition. One of the drawbacks of *non-blocking* algorithms is that the busy-waiting phase consumes CPU cycles (and therefore power) and increases contention without doing any useful work. This might reduce the application throughput when there are more threads than available hardware contexts/cores.

The absence of blocking synchronization mechanisms allows increasing performance principally because the system is more reactive. However, the implementation of *non-blocking* algorithms may require atomic operations, so that no intermediate states can be seen by other executing threads while operating on the concurrent data structure. Some of the atomic operations employed having different execution costs and complexity are *Test-And-Set* (TAS), *Fetch-And-Add* (FAA), *Compare-And-Swap* (CAS), and *Load Linked/Store Conditional* (LL/SC) [291]. Basically, these operations atomically combine a *load* and a *store* operations. At hardware level, almost all current multi-core micro-architectures already provide a set of atomic operations and memory fences. At software level, atomic operations are natively implemented by some programming languages (e.g., modern C++) and concurrent libraries (e.g., ConcurrencyKit [42]).

Lock-free and wait-free progress

An interesting and widely studied class of *non-blocking* algorithms are the ones classified as *lock-free* [80]. This term refers to the fact that the failure or the suspension of a thread in any arbitrary point during its execution cannot prevent at least one thread in the system to make progress [252]. As an example, a spin-lock based queue is not lock-free because if a thread fails after lock acquisition, no other threads would be able to complete their operations on the queue correctly. In a lock-free concurrent queue the producer and the consumer threads are able to push and pop elements concurrently by working on different positions of the queue. This does not necessarily mean that threads do not have to take into account their mutual interference. For instance, a thread may start a push/pop operation and, if the queue state changes due to the access by another thread, this must be detected and the operation restarted until it is correctly executed. Such actions (occurring in while-loops) may consume CPU cycles and power although the degree of concurrency inside the queue's code is maximized.

A stronger progress guarantee than lock-freedom is provided by *wait-free* algorithm. A concurrent algorithm is *wait-free* if it is ensured that all operations are guaranteed to complete in a finite number of steps, regardless of the timing behavior of other operations. The progress conditions enforced by lock-free and wait-free algorithms are strong enough to preclude the use of blocking constructs.

However, it is worth to remark that, lock/wait-free data structures are significantly more complicated to design and implement and consequently to verify their correctness compared to lock-based structures [140].

Performance and power consumption implications

While lock-free and wait-free algorithms are mainly studied for their progress guarantees, they are also employed for their higher throughput and lower latency [252, 197]. Furthermore, by avoiding the threads to be de-scheduled in the synchronization phases, they contribute to reducing the so-called *OS noise* which may be a source of scalability problems in many high-performance applications [254]. Unfortunately, as stated the *non-blocking* approach is not power-efficient due to the busy-waiting loop executed when a given operation cannot be immediately concluded (e.g., CAS retry loop). Although several approaches have been proposed to reduce the power consumption during busy-waiting loops, for example using *pause*, *memory barriers* or *monitor/mwait* instructions [153], none of them proved completely successful on multi-cores and busy-waiting is *de facto* considered not power efficient.

A way to improve power saving is to delay the busy-waiting phases with short phases of passive waiting obtained by executing micro-sleep system calls. This technique is called *backoff* and has been widely adopted in the implementation of spin-locks [3]. Theoretically, if the micro-sleep phases are perfectly regulated each time by an oracle, the achievable performance/power trade-off can be optimal (same performance of the pure busy-waiting approach with almost the same power saving of passive waiting). However, this is unlikely in real situations and the use of wrong values could have dramatic effects on the reactivity of the system or on its power consumption. Furthermore, the tuning phase can hardly be automatized and the solutions are in general less portable since the sleeping intervals should be regulated for each machine and application, and it might be even impossible to use sufficiently fine-grained sleep time in some OSs[1].

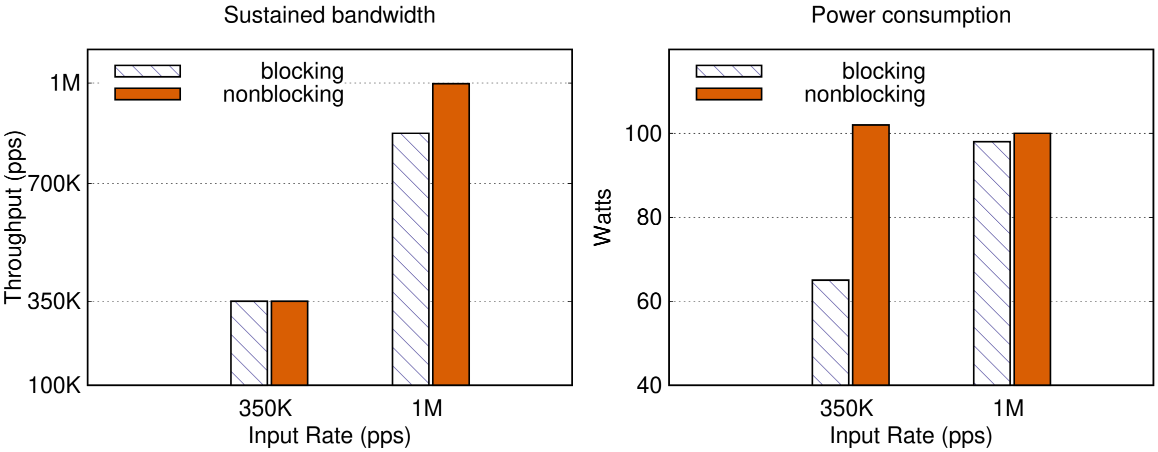


Figure 6-1: Throughput (packets per second – pps) and power consumption (Watts) of the *blocking vs non-blocking* concurrency control strategies in a pipeline application with two distinct input rates.

Figure 6-1 shows the results of the analysis we performed on a simple FastFlow pipeline application where a continuous flow of data packets is analyzed in real-time. The application will be described with more detail in Section 6.3.1. We tested the application with two different input rates: the first is of 350Kpps (three hundred fifty thousand packets per second) while the second has 1Mpps (one million packets per second). In both cases, we collected the throughput and power consumption measures obtained by two configurations of the application, the first using a *blocking* concurrency mode, and the second using a *non-blocking* mode. With the slowest input rate, the two versions achieve almost the same throughput, whereas the power consumption is significantly in favor of the *blocking* concurrency mode. This is because in that case the application is fast enough to sustain the rate, and during idle period the thread is suspended thus not consuming power. The suspension and wake-up overhead are negligible compared with the packet inter-arrival time. In the second case, the pressure to the system is higher. A packet is producer every $1\mu s$ and on average there are always packets to be processed for each stage in the pipeline. Hence, since there are no idle times, the two versions have comparable power consumption. However, the extra-overhead of the *blocking* version introduced by the protocol for managing

[1] For example, on Windows OSs is difficult to put one thread to sleep for less than one millisecond.

suspension and wake up of threads, produces negative effects on the throughput. In the *non-blocking* version no extra overhead is present thus a slight improvement of the thread processing speed allows the system to be able to sustain the arrival rate.

This simple analysis allows us to state that there is no "one size fits all" solution for the management of concurrency control when both maximum performance and minimal power consumption have to be addressed. The solution we propose consists in providing both mechanisms at the RTS level and also to provide suitable mechanisms that enable the switching of the concurrency modes of the channel between *blocking* and *non-blocking* (and vice versa) according to the actual properties of the incoming workload. This switching is transparent to the user. As we shall see in Section 6.4, a simple algorithm requiring minimal tuning can be used to implement automatic switching of the concurrency modes [314].

6.3 FastFlow channels

Communication channels are the basic mechanisms used to transfer data between different FastFlow building blocks. On shared memory platforms, FastFlow channels do not carry plain data, but references to data allocated into the shared address space. Communication channels are used as synchronization mechanisms between two distinct FastFlow nodes. The philosophy underneath FastFlow communication channels can be summarized by borrowing the *Go* language motto: "*Don't communicate by sharing memory; share memory by communicating*"[2]. In a nutshell, this means that the programmer should limit to the bare minimum or avoid at all inter-thread communications by using synchronization primitives through shared memory such as locks or mutexes, instead message-passing between concurrent entities has to be used to synchronize accesses to shared variables.

In the FastFlow programming model, the semantics of passing a reference over a FastFlow channel is that of transferring the ownership of the value pointed from the

[2] Codewalk: Share Memory By Communicating `https://golang.org/doc/codewalk/sharemem/`.

sender to the receiver. This means that the receiver is expected to have exclusive access to the pointed data structure (that could be either a partition or the entire data structure). This semantics is not statically enforced by the library, though it is the "recommended" way to use the FastFlow memory channels.

In FastFlow, all communication channels connecting two building blocks are implemented employing Single-Producer Single-Consumer (SPSC) FIFO-ordered queues [19]. The reasons behind this choice are twofold:

- Independent point-to-point FIFO channels between distinct building blocks allow increasing the control on the receive operations and on output message routing.
- SPSC FIFO queues can be implemented efficiently on systems with shared-cache [165], specifically they can be implemented without using any atomic operations (e.g., *Compare-And-Swap*).

The usage of independent FIFO channels other than facilitating the management of input non-determinism for a node allows leveraging on specific low-level features of modern many-cores such as the use of specialized NoC [67] and the user-level management of the cache-coherent protocol [66]. Moreover, in FastFlow multi-producer and multi-consumer channels are implemented by using independent SPSC channels and mediator nodes (i.e. multi-input and multi-output nodes such as the Emitter and the Collector in a farm) allowing to clearly decouple data and concurrency management of the data structure.

Concerning performance, the efficiency of synchronization mechanisms is of foremost importance in all parallel application and specifically for fine-grained ones, where the cost associated to the execution of the business logic code has about the same order of magnitude as the synchronization cost. Typically the overhead associated to synchronization has an increasingly significant effect on performance with increasing parallelism degree and decreasing synchronization granularity. In this respect, mutual exclusion using lock/unlock, is widely considered excessively demanding for high-frequency synchronizations [6, 264]. Fixed-size SPSC queues are simple to implement through circular buffers, and they perform very well on multi-cores when the producer and the consumer work on different cache lines [165]. Leslie Lamport proved that, under the Sequential Consistency (SC) memory model [1], a SPSC buffer can be implemented using only read and write operations [216]. Lamport's circular buffer is a wait-free algorithm, i.e. it is guaranteed to complete after a finite number of steps, regardless of the timing behavior of other operations. Unfortunately, the algorithm is no longer correct if the SC requirement is relaxed. This happens, for example, in those memory models where two distinct writes at different memory locations may be executed out of program order (as in the Weak Ordering memory model). A few modifications to the basic Lamport's circular buffer algorithm allow correct execution even under weakly ordered memory consistency models. Such modifications have been presented first and proved formally correct by Higham and Kavalsh [193]. The idea behind the Higham and Kavalsh SPSC queue (referred as P_1C_1) basically consists in tightly coupling control and data information into a single buffer operation by extending the data domain with a new value "−" which cannot be inserted into the queue. The special value can be used to denote an empty cell, and then used to check if the queue is empty or full without directly comparing the indexes of the queue's head and tail. The same idea has been used by Giacomoni et al. [165] by implementing an SPSC queue containing pointers to data and using as special value the "NULL" pointer. They studied the behavior of the queue in cache-coherent multi-core systems proposing a new technique called *temporal slipping* to reduce cache misses and increasing the overall performance of a queue's operations. Besides, they proved that under weakly ordered memory model, a single *Write Memory Barrier* (WMB) is sufficient to enforce completion of the data write by the producer before the data pointer is passed to the consumer.

Since, unbounded queues are mostly preferred to avoid deadlock issues in complex network topologies containing cycles without introducing heavy communication protocols, we implemented an *unbounded* lock-free SPSC FIFO-ordered queue [312, 19]. The implementation leverages both on bounded SPSC queue as implemented by Giacomoni et al. (without the temporal slipping feature) and on a simple unbounded list-based SPSC queue derived from the well-know two-lock Multi-Producer/Multi-Consumer queue proposed by Michael and Scott [249].

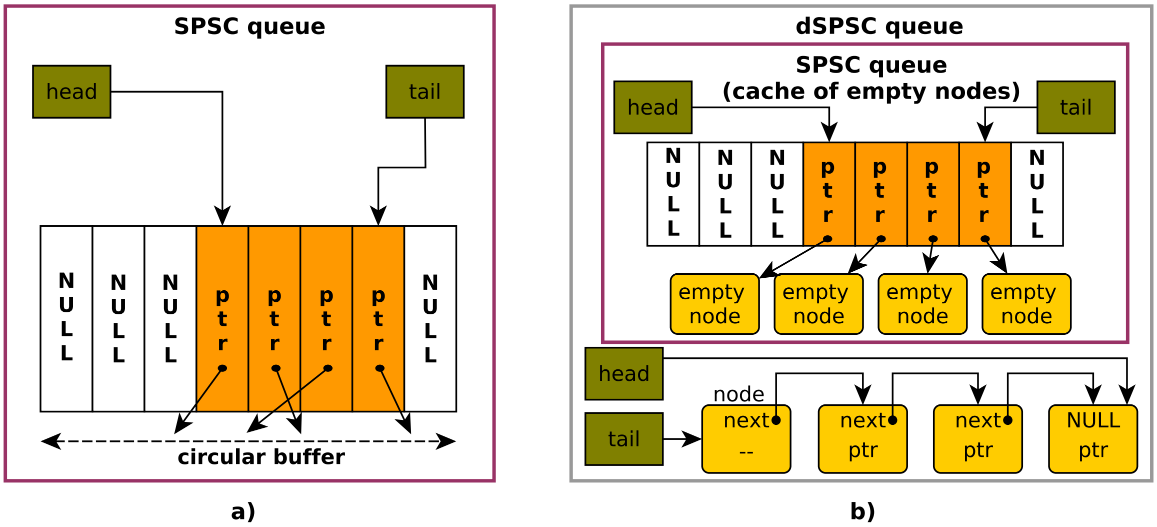


Figure 6-2: Left:) Internal structure of FastFlow SPSC bounded queue; Right:) dSPSC list-based unbounded queue with its internal cache implemented by using a bounded SPSC queue.

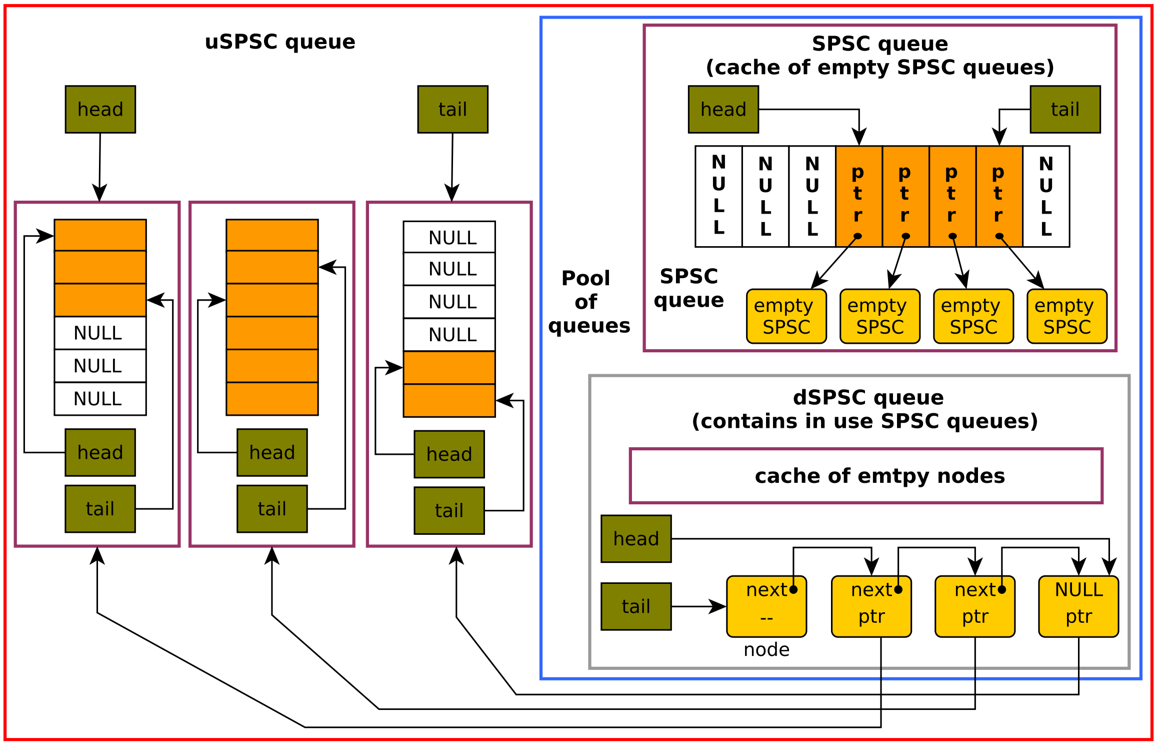


Figure 6-3: Internal structure of FastFlow uSPSC unbounded FIFO queue used to implement communication channels.

One of the main problems with list-based queues is the overhead associated with dynamic memory allocation/deallocation of internal data structures. To mitigate the overhead, it is common to use a data structure as cache, where elements are kept for future fast reuse, instead of being deallocated [189].

In the producer-consumer pattern, the producer only allocates queue elements while the consumer only frees them. To take advantage of this pattern, we used a bounded SPSC queue for implementing the internal cache of the list-based queue thus reducing the memory allocation/deallocation overheads. This optimization moves allocation overhead outside the critical path at the steady state. The resulting algorithm is called dSPSC and its main components are shown in Figure 6-2.

The unbounded SPSC queue implementation (called uSPSC) uses a pool of SPSC queues connected together. FIFO ordering is guaranteed by a dSPSC queue which contains pointer to SPSC queues that are currently in use. The pool makes use of a cache of empty SPSC queues to decrease dynamic memory allocation/deallocation. Figure 6-3 shows the internal structure of the uSPSC implementation.

The unbounded queue has the same interface as the SPSC and it can be configured to behave as a fixed-size SPSC queue by setting a configuration parameter. This simplifies the configuration of channels with unbounded capacity, as for the case of *feedback* channels that are used to route back messages to some previous building block.

In Code 11 is sketched the algorithm of the *push* and *pop* methods of the uSPSC queue together with the internal pool implementing the cache of bounded SPSC queues. The idea underpinning the uSPSC queue implementation is the nesting and composition (realized through the pool) of two well-known and efficient SPSC queues: the dSPSC and the SPSC queues. The pool of SPSC bounded queues (called also buffers) aims to minimize the impact of dynamic memory allocation/deallocation by using a fixed-size SPSC queue as a freelist. The uSPSC uses two internal pointers: `buf_w` which points to the writer's buffer (i.e. the "tail" pointer), and `buf_r` which points to the reader's buffer (i.e. the "head" pointer). Initially both `buf_w` and `buf_r` point to the same SPSC queue. The *push* method (line 1) works as follows: the

```
1  bool push(void* data} {
2    if (buf_w->isFull())
3     buf_w = pool.next_spsc_w();
4    buf_w->push(data);
5    return true;
6  }
7  bool pop(void** data) {
8   if (buf_r->isEmpty()) {
9    if (buf_r == buf_w)
10    eturn false;
11   if (buf_r->isEmpty()) {
12    SPSC* tmp=pool.next_spsc_r();
13    if (tmp) {
14     pool.release(buf_r);
15     buf_r = tmp;
16    }
17   }
18  }
19  return buf_r->pop(data);
20 }
```

```
1  struct Pool {
2   size_t size = N;
3   dSPSC  inuse;
4   SPSC   cache;
5
6   SPSC* next_spsc_w() {
7    SPSC* next;
8    if (!cache.pop(&next))
9      next = allocateSPSC(size);
10   inuse.push(next);
11   return next;
12  }
13  SPSC* next_spsc_r() {
14   SPSC* next;
15   if (inuse.pop(&next))
16     return next;
17   return NULL;
18  }
19  void release(SPSC* spsc) {
20   spsc->reset();
21   if (!cache.push(spsc))
22     deallocateSPSC(spsc);
23  }
```

Code 11: uSPSC queue implementation algorithm.

producer first checks whether the current buffer is not full (line 2), and then pushes the data into the queue. If the current buffer is full, it asks the pool for a new buffer (line 3), adjusts the buf_w pointer and pushes the data into the new buffer. The *pop* method (line 7), called only by the consumer, first checks whether the current buffer is not empty and if so pops data from the queue. If the current buffer is empty, there are two possibilities: a) there are no items to consume, i.e. the unbounded queue is really empty; b) the current buffer is empty (i.e. the one pointed by buf_r), but there may be some items in the next buffer. If the buffer is empty for the consumer, it tries to switch to a new buffer releasing the current one to be recycled by the pool (lines 12– 15). From the consumer viewpoint, the queue is really empty when the current buffer is empty and both the read and write pointers point to the same buffer. If the read and writer queue pointers differ, the consumer has to check again the current queue emptiness because between the execution of instructions at line 8 and line 9 the producer could have written some new elements into the current buffer before switching to a new one. This is the most subtle condition, in fact, if the consumer switches to the next buffer while the previous one is not really empty, a data loss will occur. We demonstrated that this condition cannot happen (the proof has been presented in [19]) even on systems with weak ordering memory models, thus the queue implementation is correct.

In the implementation of FastFlow channels, when the default *non-blocking* concurrency control mode is used, between two distinct pop retries (i.e. when `pop` returns `false`) a very small (configurable) active backoff of a few hundred cycles is executed to reduce cache pressure. It is worth noting that, even though the push method can never return *false* by definition of unbounded queue, to keep the same interface of the SPSC buffer the function returns a boolean.

6.3.1 Latency and Throughput Evaluation

In this section we analyze the base performance of the bounded and unbounded queues used to implement the FastFlow communication channels. Then, by means of two simple benchmarks we measure latency and throughput in different configurations.

Latency benchmark. The first test is a Producer-Consumer benchmark in which the first thread (P) pushes 10M data elements (each data element is just a memory pointer) into a uSPSC queue and the second thread (C) pops out tasks from the queue and performs a simple consistency check. Neither additional memory operations nor additional computation are executed. With this simple test we are able to measure the raw latency of a queue operation by computing the average value of 100 runs. We fixed the buffer size for the bounded configuration of the queue as well as for the cache size for the dSPSC queue to 1024 slots.

We tested several cases that differ in terms of the physical mapping of the two threads among the available cores. The objective is to measure the latency of operations where the two threads share different levels of caches. Therefore we consider

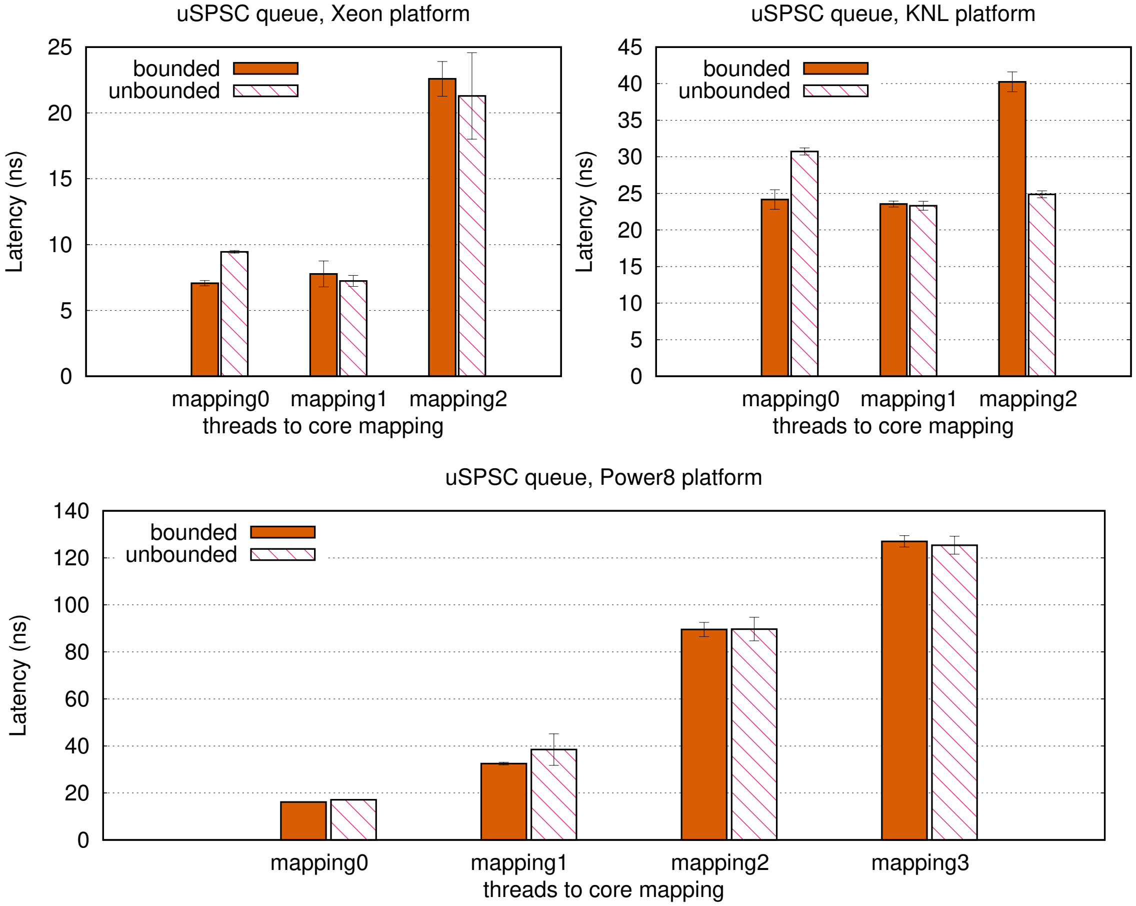


Figure 6-4: Latency (nanoseconds) of the uSPSC queue in bounded and unbounded configurations on three different platforms. *Best case scenario*: the Producer and the Consumer work on different cache-lines.

the cases where P and C are pinned to:

- the same physical core but on different HW contexts (*mapping0*);
- the same CPU but on different physical cores (*mapping1*);
- two different cores of two distinct CPUs (*mapping2* for the *Xeon* and *KNL* platforms, and *mapping3* for the *Power8* platform);
- the same CPU but two distinct CMPs (*mapping2* for the *Power8* platform only)

In Figure 6-4 are shown the average results obtained by executing 100 runs of the benchmark where P produces at maximum speed. The bounded and unbounded configurations obtain almost the same results. On the Xeon platform the latency is

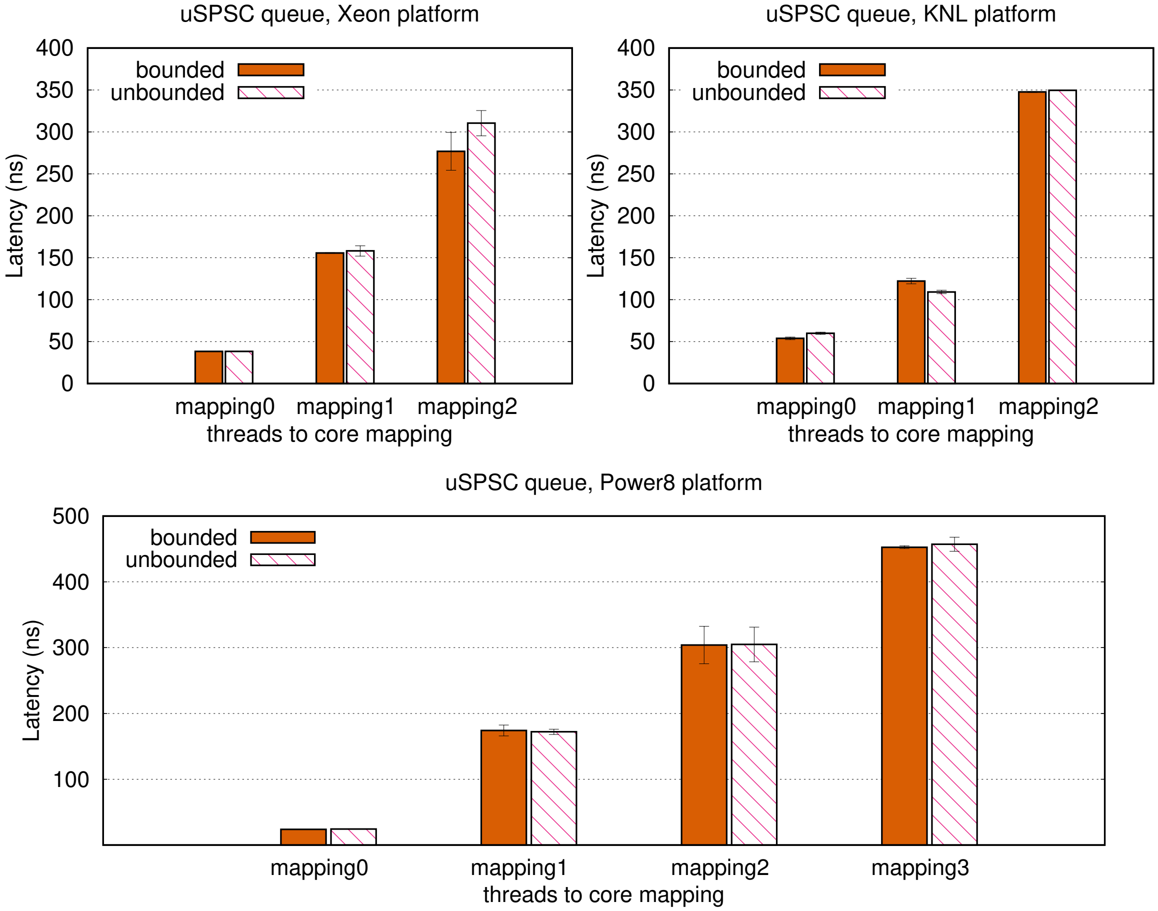


Figure 6-5: Latency (nanoseconds) of the uSPSC queue in bounded and unbounded configurations on three different platforms. *Worst case scenario*: the queue is always empty.

always lower than 25ns whereas there is a significant distance in terms of latency for the Power8 platform depending on the mapping of the threads.

In this test, P and C mostly work on distinct cache-lines (as C is a bit slower than P due to the consistency checks) and therefore the number of cache compulsory misses is minimized (on average). Thus each time the Consumer reads the head of the queue to pop out data, it reads an entire cache-line containing up to 8 pointers to data elements (typical cache-line size is 64bytes). From the perspective of reducing message latency, this is a *best case* scenario, and it represents a typical case of pipeline computations when P and C have almost the same execution time, and the execution of C is shifted ahead by a small amount of time.

To also evaluate a *worst-case* scenario, we evaluated the case when the queue is always empty, for example, because the Consumer is faster than the Producer. Since P and C work on the same cache-line, the queue accesses cause the cache-line containing the buffer entry being accessed to bounce between the last level caches of the cores hosting the threads. This phenomenon is known as *cache thrashing* which introduces extra time due to the cache coherence traffic. Such extra time penalty introduced by the cache coherence traffic for every single operation, produces up to one order of magnitude higher latency than the previous case (see Figure 6-5). However, it is worth noting that the point-to-point message latency is on average quite lower than half microsecond in the worst case, thus allowing to target fine-grained computations of the order of a few microseconds.

Scalability benchmark. To test scalability of the FastFlow channels when they are used in a *pipeline* building block we used a simple synthetic microkernel. We consider a pipeline of N sequential nodes with a feedback channel between the last and the first stage. Channels use uSPSC queues in the unbounded configuration and the number of messages injected into the pipeline is $M = 10$ million. The schema of the benchmark is sketched in Figure 6-6.

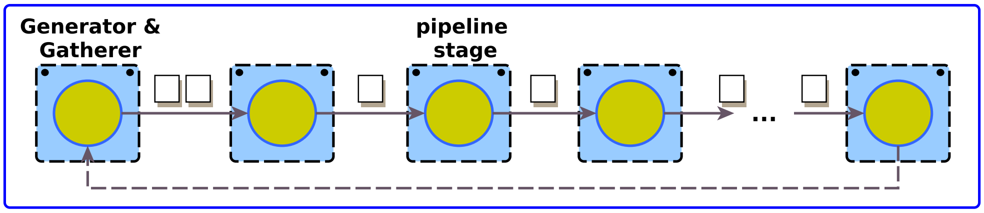


Figure 6-6: Ring benchmark: a pipeline of N sequential building blocks with a feedback channel.

The first node emits a number of messages (a first large batch of 2048 messages followed by many small batches of 128 messages) which flow around the ring. A message is just a pointer obtained from the dynamic allocation of small segments of memory (a random value between 32 and 256 bytes). The other nodes accept messages, perform basic integrity verification and pass the reference to the message to the next node. When the message returns to the first node of the ring, its memory will be deallocated. The benchmark terminates when all messages are received back

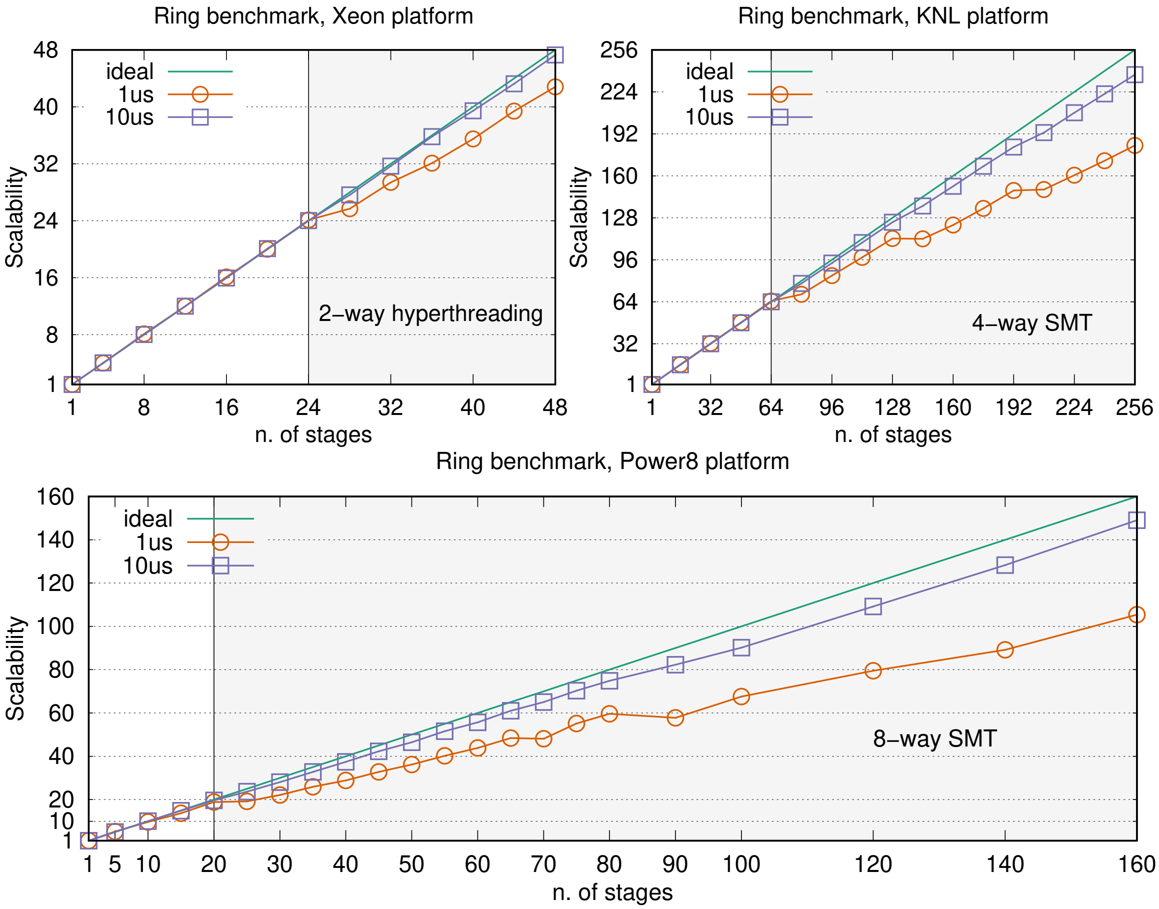


Figure 6-7: Scalability of the ring benchmark on Xeon, KNL and Power8 platforms.

by the first node. Each stage of the pipeline spends around $1\mu s$ absolute time spinning on a local variable before sending the packet to the next stage. Each thread associated to a sequential building block of the ring is statically pinned to a core whose id is the same as the thread id. For the *Xeon* and *KNL* platforms, the core ids are linearly distributed, so core id and node id correspond. For the *Power8* platform, which has a more complex hierarchy, the mapping of threads to cores is performed by first filling the first context of each physical core, then the second context and so on. Since the *Power8* has 8 contexts and 20 cores, the mapping of the node with id $i \in (1..N)$ is $x + y - 8$ where $x = 8i \; mod \; 160$ and $y = 8i \; div \; 160$. This is not necessarily the best mapping for the benchmark considered, but it is semantically equivalent to the mappings used for the *Xeon* and *KNL* platforms.

To compute the scalability of the system, we measure the system throughput considering the total amount of messages exchanged during the benchmark execution time (i.e. $(M * N)/ExecutionTime$) and we compare with the throughput obtained by a pipeline of a single node.

The scalability figures for the three platforms considered are reported in Figure 6-7. The scalability is linear up to the point when all cores are filled with at least one thread, and then it diverges from the ideal value in different ways depending on the platform considered, still reaching good maximum values in all three cases. The maximum scalability for the $1\mu s$ case are: 42.8, 183.16 and 105.4 for the *Xeon*, *KNL* and *Power8* platforms, respectively.

6.3.2 Blocking vs Non-Blocking vs Backoff

In this section we study three different concurrency control strategies: *blocking*, *non-blocking* and *non-blocking with backoff* (simply *backoff* in the following). For evaluating the performance and power implications of using different concurrency control mechanisms we consider two benchmarks studying both latency and throughput

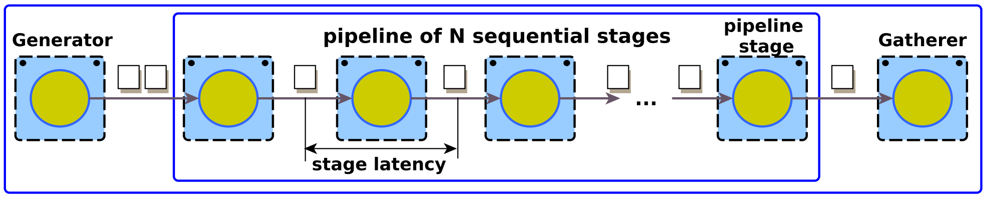


Figure 6-8: Latency benchmark: a pipeline of $N + 2$ sequential building blocks.

The first benchmark is a linear pipeline of N sequential nodes where the first stage (the *Generator*) injects into the chain a continuous stream of packets at a predefined constant rate for a given amount of time (30 seconds in our tests). A *Gatherer* stage collects all packets injected into the pipeline (see Figure 6-8). It is worth to remark that, the latency benchmark we consider in this section is different from the one we used for the evaluation of the cost of push/pop operations for the uSPSC queue presented in Section 6.3.1. Here the objective is to measure the average latency of a message for crossing a single pipeline stage (i.e. the average time between the push of a message by the previous stage and a pop operation by the next stage for receiving the message).

The second benchmark is the ring benchmark we already described in Section 6.3.1 (see also Figure 6-6).

Latency evaluation. The latency in the *pipeline benchmark* is computed by starting a timer for each packet before sending it into the chain and stopping the timer when the packet is received by the *Gatherer* stage. The *Gatherer* computes a moving average with an overlapping constant size of 10 values and eventually produce in output the average value divided by the $(N + 1)$ (i.e the number of channels of the pipeline). The overall throughput in the *ring benchmark* is computed considering the total amount of messages exchanged during the benchmark execution time and precisely $M * (N + 1)/TotalExecutionTime$ where M and N are the number of messages and the number of stages, respectively.

We studied three different configurations:

1. *non-blocking* concurrency mode where all threads continuously keep polling their input/output queues with a minimal active backoff of a few hundred CPU cycles;
2. *blocking* concurrency mode where all threads are immediately put to sleep waiting for a wake-up signal if their input/output queues are empty/full;
3. *backoff* where all threads use different retry policy strategies alternating active polling and micro sleeping periods.

For the *backoff* configuration, we consider two distinct cases:

(i) *backoff-2l* having two levels of retry policies: *aggressive* and *moderate*;

(ii) *backoff-3l* having three distinct levels of retry policy: *aggressive*, *moderate* and *relaxed*.

In the *backoff-2l* the *aggressive* policy performs 256 active polling attempts then it switches to *moderate* strategy where, between two distinct retries, the polling thread sleeps for 1 millisecond. In the *backoff-3l* case, the *aggressive* policy performs 64 attempts of active polling then it switches to *moderate* where the thread performs 256 attempts and between two distinct retries it sleeps for 50 microseconds then, if

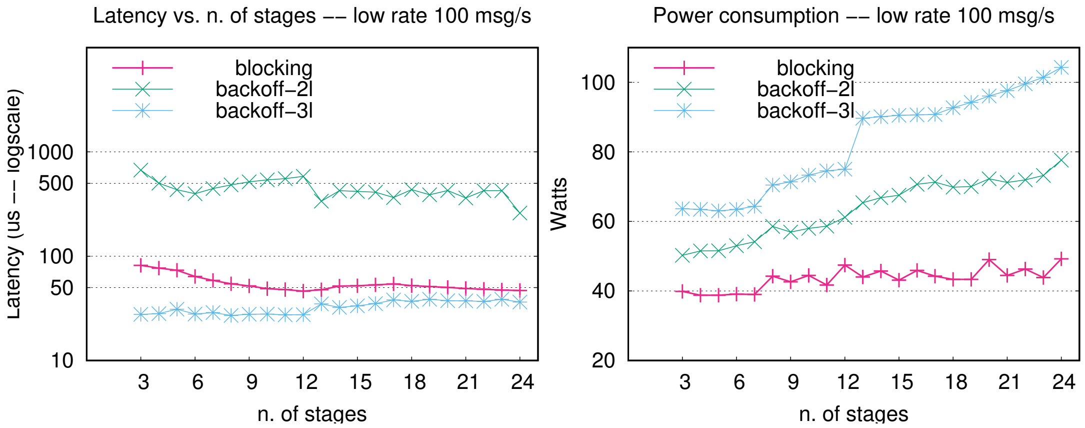


Figure 6-9: Latency and power consumption of the pipeline benchmark in *blocking* and *backoff* configurations for "low" message rate (100msg/s).

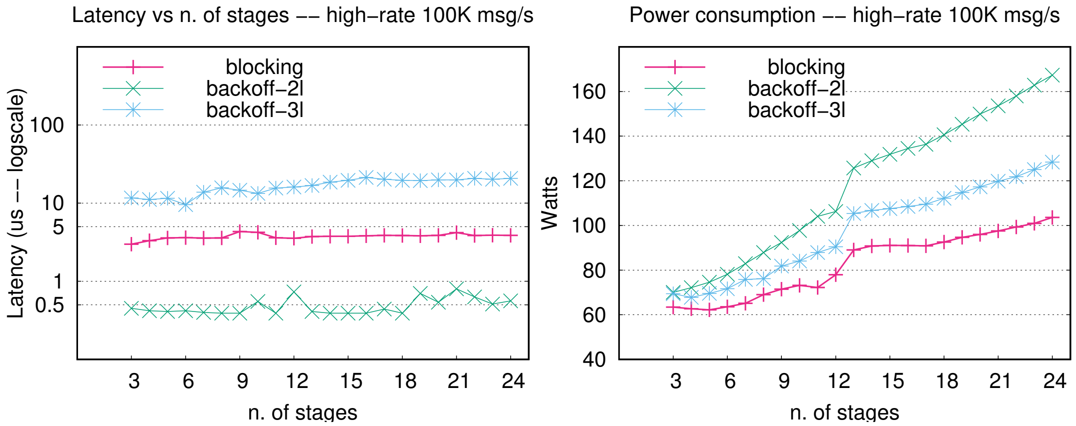


Figure 6-10: Latency and power consumption of the pipeline benchmark in *blocking* and *backoff* configurations for "high" message rate (100Kmsg/s).

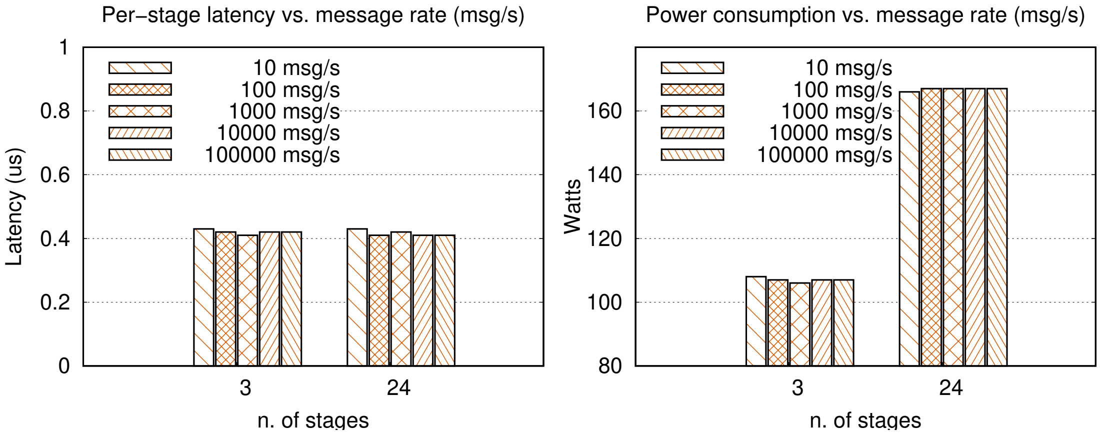


Figure 6-11: Latency and power consumption varying the message rate for the pipeline benchmark in the *non-blocking* configuration.

the queue is still empty/full, it switches again moving to the *relaxed* mode where the thread sleeps for 1 millisecond between two distinct retries. The *backoff-2l* is characterized by a more aggressive behavior where the thread spins or sleeps, whereas the *backoff-3l* more gradually moves from spinning to sleeping over time. We select these specific values for the sleeping time and for the number of retries since they provide a good trade-off between power consumption and performance. To measure the power consumption, we used the MAMMUT[3] library [132], that on the multi-core system used for the tests relies on RAPL counters.

Figure 6-9 shows the results of the *pipeline benchmark* in the *blocking* and *backoff* configurations for low message rate (10msg/s). For such low rate the best average latency is obtained using the *backoff-3l* configuration which has a *moderate* policy that performs small sleeps (50 microseconds) whereas the *backoff-2l* has a longer sleeping period (1 millisecond). It is worth pointing out that increasing the number of stages the *blocking* and *backoff-3l* case have closer average latencies. Concerning power consumption, as expected, the *blocking* concurrency mode is the most power efficient consuming less than 50 Watts in all tests.

The results obtained with higher input message rage (100K msgs/s) are reported in Figure 6-10. In this test, a packet is produced each 10 microseconds. The best latency is obtained by the *backoff-2l* configuration while the *backoff-3l* is the worst configuration. This can be explained considering that the *backoff-2l* has a higher number of retry attempts in the *aggressive* strategy. In fact, it performs 256 retries of active polling instead of 64 retries of the *backoff-3l*. Therefore, with the given rate, in the *backoff-3l* the spinning thread does enter the retry loop of the *moderate* strategy which between two consecutive retries the thread is put to sleep for at least 50 microseconds which is a time higher than the actual rate. Conversely, the *backoff-2l* is the most power-hungry strategy consuming in the 24 stages configuration 167 Watts.

In Figure 6-11 we show the average latency and the power consumption for the *pipeline benchmark* using the *non-blocking* configuration, varying the message rate

[3] `http://danieledesensi.github.io/mammut/`

between 10msg/s to 100Kmsg/s. We considered two configurations for the pipeline: 3 and 24 stages. As can be seen, the latency is almost constant (about 400 nanoseconds) for the two cases regardless of the input message rate and the number of stages of the pipeline chain. The same applies for the power consumption that is constant regardless of the input rate. The difference between the case with 3 and 24 pipeline stages is due to number of CPUs used in the two configurations, in fact each CPU of the platform considered for the tests hosts 12 cores. Therefore, in the pipeline configuration with 3 stages only the cores of the first CPU are used, allowing the OS to put the second CPU in low-power mode.

Throughput and CPU utilization. Concerning the throughput, in Figure 6-12 is shown the results obtained by the *non-blocking*, *blocking* and *backoff-2l* configurations. As expected, the *blocking* concurrency mode offers the lowest overall system throughput due to the higher overhead of the concurrency mechanisms, whereas the *non-blocking* and the *backoff-2l* configurations have exactly the same system throughput reaching a maximum value of about 55 Mpps (millions-packets-per-second).

Finally, in Figure 6-13 is reported the average CPU utilization of the *pipeline benchmark* for the case with 24 stages. As can be observed, the *non-blocking* configuration uses almost 100% of the CPU cycles available regardless the input rate, the *blocking* configuration uses less than 1% of CPU cycles for low message rate (100 msg/s) and about 10% CPU cycles for high message rate. The *backoff* configurations consume much less of the *non-blocking* concurrency mode for low message rate while for high message rate the *backoff* concurrency mode consumes about 95% and 40% CPUs cycles in the *backoff-2l* and *backoff-3l* configurations, respectively.

The results obtained demonstrate that the *non-blocking* concurrency model is the most efficient and stable one when considering latency reduction as the most important metric. On the other hand, it is also the most power expensive both in terms of CPUs cycles and in terms of power consumed. The *backoff* and *blocking* concurrency models offer different performance results on the basis of the given input rate. Moreover, the tuning of the backoff's sleeping time and the number of attempts

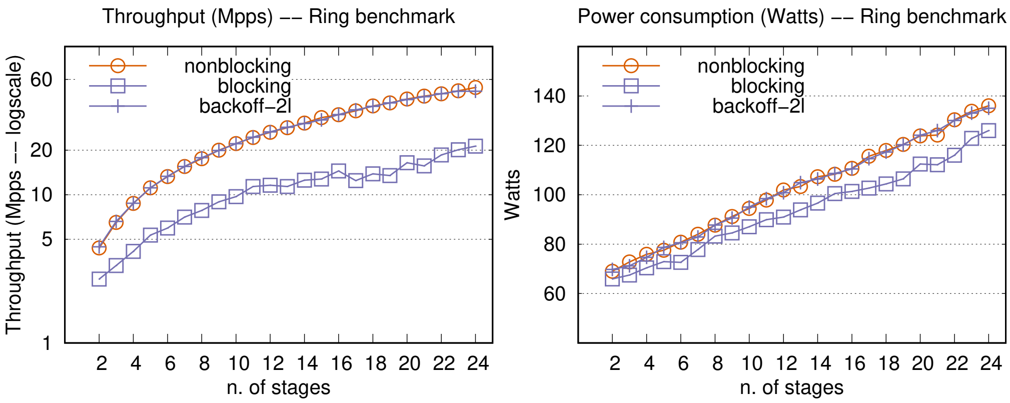


Figure 6-12: Throughput and power consumption of the ring benchmark.

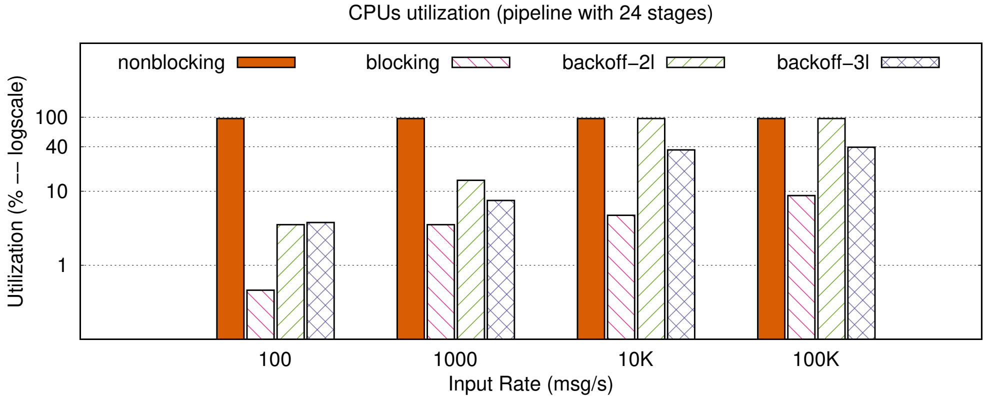


Figure 6-13: CPUs utilization (% – logarithmic scale) for the pipeline benchmark with 24 stages considering 100,1000 and $100K$ msg/s.

for each strategy, is not an easy task and need to be regulated on the basis of the rate. The *backoff* strategy is a good compromise if the most important optimization metric is the throughput, in fact it can offer the same performance as the *non-blocking* strategy at high rate and to offer almost the same power consumption as the *blocking* strategy for low input rate. When latency is not the primary metric to optimize, the *blocking* concurrency model offers the most efficient and effective solution providing a good balance between absolute performance and power consumption.

6.4 Automatic Concurrency Control

The FastFlow framework supports both *non-blocking* and *blocking* concurrency control mode for accessing communication channels. The *non-blocking* mode promotes system reactivity whereas the *blocking* mode promotes both power saving and more fair execution of threads on a limited set of cores.

Before the program starts it is possible to switch between *non-blocking* and *blocking* concurrency mode by using the method `blocking_mode(blk)` implemented in each building block. If the function parameter is set to `true`, the concurrency control mode selected is *blocking* (the default one) otherwise is *non-blocking*. If the method is called on the outer-most pipeline, the concurrency mode will be set accordingly for all internal channels contained in the pipeline. Since input and output channels of a given node are independent, it is also possible to selectively switch the concurrency mode either for the input channel or the output channel. This is an operation acting only on one side of a channel connecting two nodes (either the producer or consumer side), therefore it has to be explicitly applied also to the other side of the same channel (i.e. either to the previous or subsequent node). This is an important feature offered by the FastFlow node that allows implementing autonomic algorithms capable of configuring different parts of the FastFlow network with different concurrency control strategy according to some monitored metrics (e.g., CPU utilization, power consumption, input data rate).

As we shall see in the following, it is also possible to dynamically switch between the two concurrency mode at run-time for each building block. This requires the implementation of a protocol to synchronize the switching between two distinct nodes of the application topology. The algorithm has been described with detail in Torquati et al. [314], here we summarize the results obtained.

push and pop operations. Here we want to focus on the algorithm used for implementing both *blocking* and *non-blocking* concurrency control strategies in FastFlow. In Code 12 is reported the implementation of the *concurrency control push* and *pop* algorithms executed on the output and input channels of a node, respectively. The boolean variable `blocking_mode` holds the current status of the channel: `true` *blocking*, `false` *non-blocking*. Besides, each channel has associated a mutex and a condition variable used to implement events notification between the producer and the consumer nodes.

```
 1 void ccpush(void* data) {
 2   if (blocking_mode) {
 3     for(;;) {
 4       bool r= chQ.push(data);
 5       if (r) {
 6         signalConcumer();
 7         break;
 8       } else
 9         passiveWait(bigBackoff);
10     }
11   } else {
12     while(!chQ.push(data))
13       activeWait(smallBackoff);
14   }
15 }
```

```
 1 void ccpop(void** data) {
 2   if (blocking_mode) {
 3     for(;;) {
 4       bool r= chQ.pop(data);
 5       if (r) {
 6         signalProducer();
 7         break;
 8       } else
 9         passiveWait(bigBackoff);
10     }
11   } else {
12     while(!chQ.pop(data))
13       activeWait(smallBackoff);
14   }
15 }
```

Code 12: Push and pop operations implementing both *blocking* and *non-blocking* concurrency control strategies for accessing the FastFlow channel.

To push a message into the output channel the RTS first tries to put the data pointer into the uSPSC queue associated to the channel (called `chQ` in Figure 12). If the operation succeeds (it may fail if the queue is configured to have a limited capacity), depending on the current concurrency mode of the node, two different actions are taken. In the case of *non-blocking* mode, the operation has been successfully completed without any further action. Otherwise, in *blocking* mode, the RTS signals the consumer node to wake up if it is waiting on the condition variable associated with the channel (line 6). If the push fails and `blocking_mode` is `false` the operation is executed again until it will complete with success. In between two retries, a small amount of active waiting on a local variable is executed (line 13). The backoff value corresponds to a few hundred clock cycles and is necessary to reduce cache thrashing. If `blocking_mode` is `false`, then the run-time suspends the producer (line 9) for a finite amount of time. In this case, the sleep time is higher than the active waiting time (hundreds thousand clock cycles). Such sleep time value is not particularly critical in terms of power consumption provided it is high enough to allow the OS to put the thread to sleep (typically few hundreds of microseconds on modern OSs) [313]. The producer will be woken up either by an event signaled from the consumer that made space in the queue (i.e. after a successful `chQ.pop` operation – line 4) or because the timeout associated to the condition variable expires. In the first case, the value will be popped out from the queue, in the second case the entire cycle is executed again and the consumer may fall asleep again. The pop operation algorithm is symmetric to the push one.

It is worth pointing out that by properly changing the `blocking_mode` variable for a given channel (this operation can be done selectively for the input and output channels of a given node), it is possible to switch between *blocking* and *non-blocking* concurrency mode for a pair of nodes.

Power-Aware concurrency control. The implementation of the algorithm for automatically switching between *blocking* and *non-blocking* concurrency mode leverages a manager node that is in charge of making decisions for the building block. At configurable time intervals, by collecting monitoring information about the current performance and power consumption of the entire application, the manager decides which message queue should operate in *blocking* or *non-blocking* mode by directly notifying the producer and consumer nodes of the pipeline network.

We use the following notation. The average latency of a `ccpop` operation is L^{b}_{pop} and L^{nb}_{pop} for the *blocking* and *non-blocking* concurrency control modes, respectively. If the operation fails, the node waits for new data to arrive by suspending itself or by doing active waiting on a local variable. Let us denote this average waiting time with L_{idle}. The average time spent processing input data is denoted as L_{proc}. The average latency of a `ccpush` operation is L^{b}_{push} and L^{nb}_{push} for the two concurrency modes, respectively. For simplicity sake, we suppose that output queues for a node have an unbounded capacity, therefore the `ccpush` operation will always succeed.

The breakdown of a single loop iteration of a node is sketched in Figure 6-14. Timing values (e.g., L_{idle}, L_{proc}) are collected by each single FastFlow node and stored in its internal object variables. The manager can read such monitoring traces by directly accessing the internal state of each node without any extra synchronization.

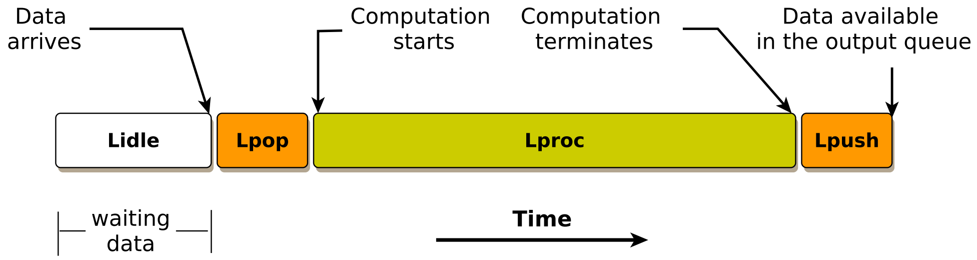


Figure 6-14: Different kinds of latencies in the FastFlow node operations.

It should be noted that the `blocking_mode` variable is an atomic variable to allow the manager node to change its value without race conditions.

From blocking to non-blocking. Suppose that when the application starts, it uses all the message queues in *blocking* mode. To improve the throughput of the application, we have to improve the throughput of its slowest node, i.e. the one with the highest latency. We call this node S. If $L_{idle}(S) > 0$, despite being the slowest node in the application, it is still fast enough to process the incoming data, so there is no need to improve the throughput of the application at all. Otherwise, we can improve the throughput of S by reducing the latency of both the `ccpop` and `ccpush` operations. Let us start with the `ccpop` operation. Switching the input queue to *non-blocking* mode would have no impact on the power consumption since $L_{idle} = 0$ and S will not do active waiting. Now let us consider the `ccpush` operation. We could switch the output queue of S to *non-blocking* mode and reduce L_{push} as well. Let us call T the successor of S. Since S is slower than T, $L_{idle}(T)$ is greater than zero. If the message queue is in *blocking* mode, while idling, T is suspended on the condition variable. However, after switching to *non-blocking* mode, T will start doing busy-waiting, thus increasing the power consumption of the application. To determine if the increase in power consumption is worth the increase in performance, we decided to let the application user (or the system developer) set some preferences for the application, by specifying maximum allowed increase in power consumption for each 1% increase in performance. Similarly, the user might just set a maximum power consumption of the system letting the RTS optimize the performance with the given power budget (this is also known as *Power Capping* [92]).

For evaluating the outcome of the decision, we adopt a *rollback-based* approach. When a potential performance improvement for the output queue is detected, the algorithm switches the queue from *blocking* to *non-blocking.* Then, the performance and the power consumption are monitored for the next time interval. If the results of the switching do not comply with the user requirements, the decision is reverted, otherwise it is kept. Since in both cases we improved S by switching its input queue, the slowest node might now be a different one. If this is the case, the algorithm is executed on the new slowest node, otherwise it terminates. To avoid too many rollback operations, if a node was involved in a rollback operation, it is marked and it is not evaluated again for a time interval that can be specified by the user.

From non-blocking to blocking. Due to workload fluctuations, the system could start receiving less data per unit of time. In such a case, the message queues will become empty and some nodes will start doing active waiting on their input queues consuming clock cycles and therefore power. By switching a queue to *blocking* mode, the nodes using the queue will start doing passive waiting sleeping for the configured amount of time. However, as we have seen in Section 6.3.2, we will also increase the latency of `ccpush` and `ccpop` operations. To ensure that this switch does not decrease the throughput of the nodes, it is sufficient to ensure that the increase in the `ccpush` and `ccpop` latency is "*absorbed*" by the idle latency, i.e. even if these operations will last longer, the nodes will still have enough time before receiving the next data element, thus not reducing their performance. To do so, it is sufficient to find the pairs of nodes P (Producer), C (Consumer) such that the following condition is true:

$$L_{idle}(C) > L^{b}_{pop}(C) - L^{nb}_{pop}(C) \textbf{ and } L_{idle}(P) > L^{b}_{push}(P) - L^{nb}_{push}(P)$$

If these conditions hold, we will have $L_{idle} > 0$ for both nodes after switching to *blocking*, thus not reducing their throughput.

6.4.1 Evaluation

In this section we validate the automatic concurrency control algorithm described in the previous section by using a real streaming applications. We conducted all tests on the *Xeon* platform (see Table. 6.1).

For implementing the manager node, we used the functionalities provided by the Nornir framework [131, 293]. It uses the mechanisms provided by the FastFlow runtime and on top of them implements the algorithm described in Section 6.4 to decide when to switch from *blocking* to *non-blocking* and vice versa. The algorithm activates once every second, and it takes just a few milliseconds to decide which queues must be switched.

To compute L^{b}_{push}, L^{nb}_{push}, L^{b}_{pop} and L^{nb}_{pop} which are needed to decide when to switch the concurrency mode, we used the same benchmark described in Section 6.3.1 considering the case of only two FastFlow nodes. On the *Xeon* platform we used the following average values: $L^{b}_{push} = 27us$, $L^{nb}_{push} = 0.4us$, $L^{b}_{pop} = 0.8us$ and $L^{nb}_{pop} = 0.01us$.

Malware Detection application. The Malware Detection application is described in our previous work [110]. From the parallel standpoint, this application is structured as a 3-stage pipeline where the middle stage computes the most expensive part of the application and can be implemented in parallel by replicating several times the same function operating on a partitioned hash table. In FastFlow, this network can be easily and efficiently implemented by using a single farm building block with a user-defined scheduling policy.

Each Worker of the task-farm, after having identified the protocol, searches for a predefined set of “signatures” (representing malware binaries) inside each HTTP packet. The packets are scheduled to one of the Workers according to the value of a key computed by the first logical stage of the pipeline, that is combined with the task-farm Emitter (see Figure 6-15). The application is implemented using the Peafowl network framework whose RTS is based on the FastFlow library [110].

In this experiment we used the *Xeon* platform. The application graph is composed by 24 nodes (22 Workers, 1 Emitter and 1 Collector nodes) The arrival rate of the

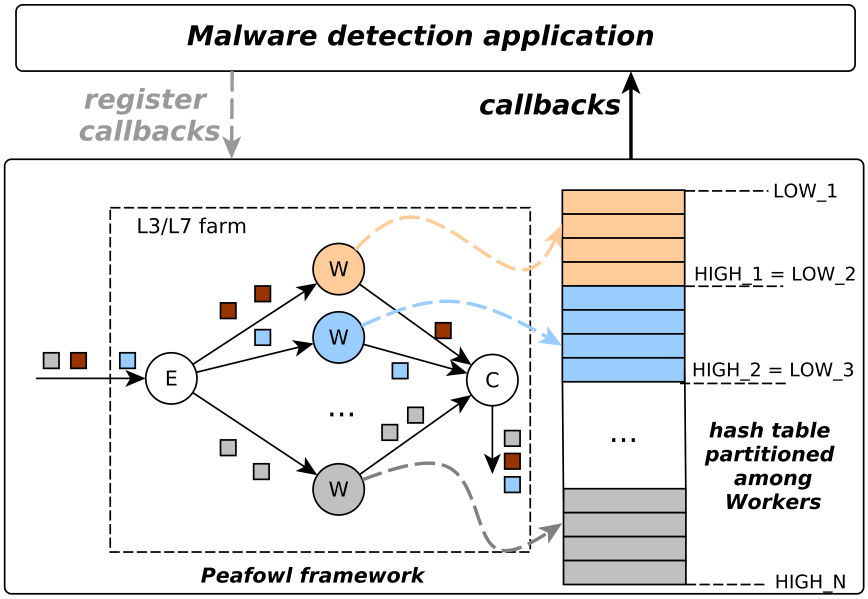


Figure 6-15: Malware detection application implemented using the Peafowl framework that leverages the FastFlow *farm* building block.

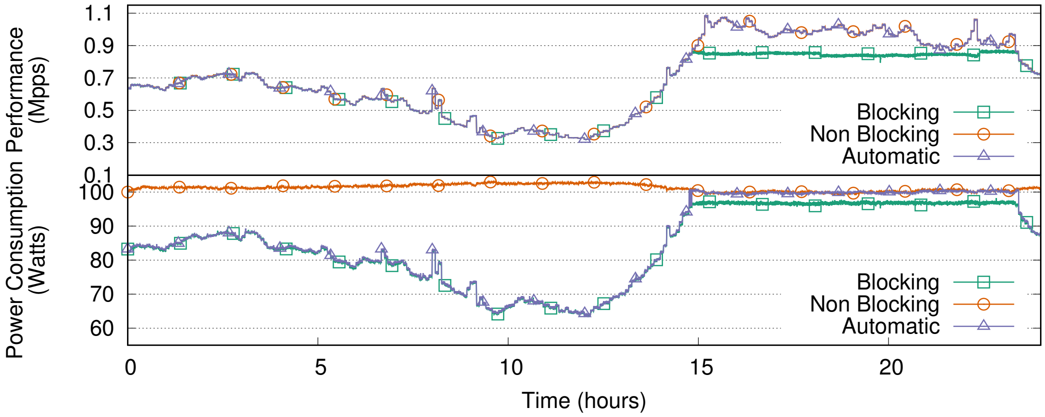


Figure 6-16: Performance and power consumption comparison of *blocking*, *non-blocking* and *automatic* concurrency control strategies for the Malware detection application.

packets to the application is variable. In our test, we used the rate that characterizes a modern *Internet Exchange Point* network[4]. For the malware detection part, we used a subset of the database used by the *ClamAV* antivirus[6], containing 2000 signatures.

[4] `https://stats.linx.net/`, (IXManchester)[5]. We scaled it down by a 3x multiplicative factor to match the maximum performance achievable on our target architecture.

[6] `https://www.clamav.net/`.

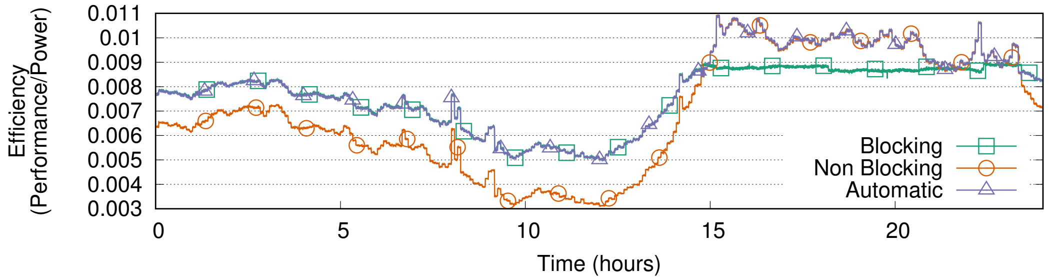


Figure 6-17: Comparison of efficiency of *blocking*, *non-blocking* and *automatic* concurrency control strategies on the Malware detection application.

The results of our test are sketched in Figure 6-16, showing that the *automatic* policy is able to achieve the maximum performance while having the same power consumption obtained by the *blocking* concurrency mode. Between 15 and 22 the *blocking* strategy has a lower power consumption but it cannot sustain the same arrival rate as the *non-blocking* one.

In Figure 6-17 we show another interpretation of the result, by plotting the efficiency of the different concurrency control techniques, expressed as the ratio between the performance and the power consumption. As we can see from the plot, the *automatic* strategy is always characterized by the highest efficiency between those of the other two techniques.

Now we consider the case when the database of signatures is more extensive than the one considered in the previous tests. This means that the generic Worker executes more work for each input packet so its service time increases. By increasing the time spent computing the single input element (i.e. L_{proc} in Figure 6-14), the relative impact of L_{pop} and L_{push} decreases, thus reducing the benefit of the *non-blocking* strategy over the *blocking* one. In a nutshell, for application characterized by a high L_{proc}, the *blocking* strategy performs as well as the *non-blocking* one.

While in the previous tests we considered a database of 2000 signatures, here we consider two other cases: a bigger database of 45000 signatures and a very large database of 90000 signatures. The results for the two cases tested are reported in Figure 6-18 in the top and bottom plot, respectively.

Moving to a larger database of signatures, the performance gap between *blocking*

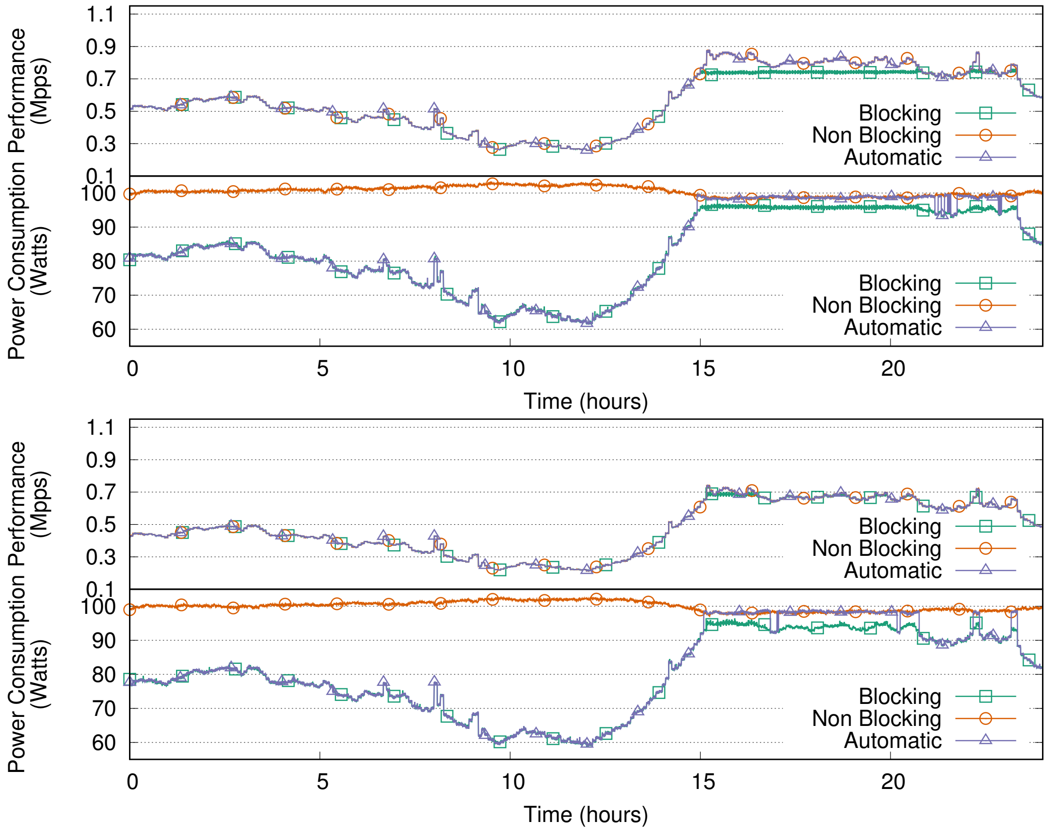


Figure 6-18: Performance and power consumption comparison of *blocking*, *non-blocking* and *automatic* concurrency control strategies for the Malware detection application when the database of signatures is of 45000 (top plot) and 90000 (bottom plot) signatures, respectively.

and *non-blocking* strategies gets closer (see the top plot of Figure 6-18). As expected, the number of packets-per-second the system can sustain over time decreases because of the increased service time of each Worker. When the largest database of signatures is considered, the two concurrency control policies provide almost the same performance (see the bottom plot of Fig. 6-18), whereas, in terms of the ratio between performance and power consumption, the *automatic* policy still provides the best efficiency.

To better quantify the relation between service time and maximum throughput of the *blocking* and *non-blocking* concurrency control strategies for the Malware Detection application, we reported the measured values in Table 6.2. Specifically, the table shows the average Worker service time (in microseconds) for the entire execution of the application, and the maximum number of packets-per-second the system can sus-

Worker's service time (μs)	**Blocking vs Non Blocking throughput difference (%)**
19	24.23%
23	15.3%
26	5.15%

Table 6.2: Performance gap between *blocking* and *non-blocking* strategies considering different Worker's service time for the Malware detection application.

tain. The lower the Worker service time, the more significant the performance gap between the two strategies. This confirms that the *non-blocking* strategy introduces lower overhead than the *blocking* one. From our tests, for service time higher than 26 microseconds, the *non-blocking* policy starts providing only marginal benefit if any at all. Therefore there is a clear threshold in the node's service time that delimits the point below which the *automatic* algorithm provides benefits and is worth to be used.

6.5 Summary

In this chapter, we presented the implementation of the FastFlow communication channel, which is the only mechanism used to transfer references to data between distinct building blocks. We tested the performance of the channels concerning point-to-point latency (both best-case and worst-case scenarios) and system throughput by considering a pipeline of multiple nodes. The results demonstrate that the FastFlow channel is capable of offering both low-latency and high-throughput on the three different multi-core platforms considered.

Besides, we studied *blocking* and *non-blocking* concurrency control strategy for regulating concurrent accesses to the communication channel. FastFlow offers both concurrency control strategies that represent the two best solutions concerning performance and power consumption for low and high data rate, respectively. The two concurrency modes are integrated into the FastFlow implementation of communication channels.

The *non-blocking* strategy is the most performing one and the most power-hungry. Its power consumption depends on the number of cores used and not on the input data rate. The *blocking* strategy instead is the most power-efficient from low to medium input rates and when more threads than physical cores are used. However, its associated overhead does not always allow the application to reach the maximum throughput, specifically for fine-grained computation.

Finally, we presented an autonomic algorithm that allows us to automatically select the best concurrency mode at run-time based on monitoring information and pre-computed measures related to the average latency of push and pop operations on the target platform.

The *automatic* strategy proposed represents a good trade-off between absolute performance and power consumption when a static-time decision between the two policies is difficult to make. It allows exploiting both benefits of the *blocking* and *non-blocking* strategies leading to optimal values of the power/performance ratio.

Chapter 7

Sequential and Parallel Building Blocks

7.1 Introduction

In this chapter we describe and assess the implementation of the FastFlow *sequential* and *parallel* building blocks. Sequential building blocks are the fundamental elements of any FastFlow streaming network and the primary components of the parallel building blocks.

In Section 7.2.1 we describe the FastFlow *node*, the basic concurrent object implementing a standard sequential building block. *Multi-input* and *multi-output* nodes are presented in Section 7.2.2, while the *sequential nodes combiner* is presented in Section 7.2.3. An evaluation of the overhead introduced by the sequential nodes combining operation is presented in Section 7.2.4. In Section 7.3 the *pipeline*, *farm* and *all-to-all* building blocks are discussed presenting their features and capabilities, pointing out the main differences between the *farm* and *all-to-all* building blocks. In Section 7.3.2 the new concurrency throttling mechanisms of the *farm* building block will be described and tested. The parallel overhead introduced and the scalability of the parallel building blocks are evaluated considering as target platforms both a state-of-the-art multi-core and the Intel Xeon Phi KNL many-core.

7.2 Sequential Building Blocks

7.2.1 The FastFlow *node*

Within the FastFlow framework, a concurrent activity, either implemented by a single thread of execution or by multiple threads, is called *node*. It represents the base C++ class for any parallel and sequential building block. The data type representing a FastFlow node is *ff_node_t*, which is a C++ abstract class containing several methods defining the data-flow behavior of the node. Three virtual methods of the *ff_node_t* class, are of particular importance:

```
virtual TOUT* svc(TIN* task) = 0;   // encapsulates user's code
virtual int   svc_init();           // initialization code
virtual void  svc_end();            // finalization code
```

Each FastFlow node must implement at least the method `svc` (which stands for *service*). It gets as input argument a pointer to a data element and returns a pointer to the same data or to another data allocated within the object. The `svc` method is called by the FastFlow RTS as soon as a data element is available to be consumed by the node, i.e. as soon as a data element is present in one of its input channels. The other two methods are automatically invoked once by the FastFlow RTS when the concurrent entity associated to the node starts (`svc_init`) and right before it terminates (`svc_end`). These virtual methods may be overwritten in the user supplied FastFlow node sub-class to implement initialization code and finalization code, respectively. If the `svc_init` returns a value different from zero, it means that something in the initialization phase went wrong and the node is terminated. While in the `svc_init` method it is possible to send data into output channels, in the `svc_end` method data produced in output will not be delivered. The three "service" methods of the FastFlow node class implement the so-called *business logic* of the node.

A standard FastFlow node has one input channel and one output channel. It processes data items delivered in its input channel and conveys the results to its output channel. Particular cases of FastFlow nodes may be implemented with no input channel or no output channel (they are often called *source* and *sink* nodes, respectively). The former is used to install a concurrent activity generating an output stream of data elements (e.g., by reading data coming from standard input, files or network sockets); the latter is used to install a concurrent activity consuming an input stream (e.g., to present results on a standard output or to store them into disk files or in a database). The FastFlow node terminates either if it receives a special data element called `EOS` (*End-Of-Stream*) from all its input channels or if it returns the `EOS` special value as a result of the `svc` method. It is possible that for a single input element no element has to be produced into the output channel, then the special value `GO_ON` must be returned by the `svc` method. This special value tells the RTS to maintain the concurrent entity associated to the node alive and to keep monitoring its input channel(s) for new data elements to process. The simplified life cycle of the generic FastFlow node is informally described in Code 13.

```
 1 do {
 2   if (svc_init() < 0) break;
 3   do {
 4     in = input_channel.pop();
 5     if (in == EOS) {
 6       // if this method has been redefined, the user's method
 7       // is called and informed that the EOS has arrived
 8       eosnotify();
 9       output_channel.push(EOS);
10     } else {
11       out = svc(in);  // it calls the business logic code
12       if (out == GO_ON) continue;
13       output_channel.push(out);
14     }
15   } while(out != EOS);
16   svc_end();
17 } while(true);
```

Code 13: Schema of the life cycle of the generic FastFlow node.

When the thread implementing the node is started, the `svc_init` method is called (line 2) and then, if a data element is present in its input channel, the element is

```
#include <ff/ff.hpp>
using namespace ff;
struct myNode:ff_node_t<int> {
  int svc_init() {
    std::cout << "Hello. I'm going to start\n";
    counter = 0;
    return 0;
  }
  int* svc(int*) {
    if (++counter > 5) return EOS;
    std::cout << "Hi! (" << counter << ")\n";
    return GO_ON; // keep calling the svc method
  }
  void svc_end() { std::cout << "Goodbye!\n"; }
  // starts the node and waits for its termination
  int  run_and_wait_end(bool=false) {
    if (run() < 0) return -1;
    return wait();
  }
  long counter;
};
int main() {
  myNode mynode;
  return mynode.run_and_wait_end();
}
```

Code 14: How to define and execute a standalone FastFlow node.

extracted from the channel (line 4) and, either the method `eosnotify` is called to notify the user's code that the EOS has arrived (8) (this requires that the user has overwritten this method), or the `svc` user's function is called passing the data element received (line 11). If the input channel is empty, the run-time waits until a valid data element is present in the input channel according to the concurrency control mode selected for the node (see Section 6.2). In some cases, it is useful that for a single input element more than one element has to be produced to the output channel of a node. To this end, the method `ff_send_out` can be used inside the `svc` method to produce data elements into the node's output channel. An example of how to use `ff_send_out` is shown in Section 7.2.3. In Code 14 we show a very basic example of how to define a sequential FastFlow node. As we shall see, usually, FastFlow nodes are not used alone, but they are added to a pipeline building block that connects them in the proper order and then starts their execution. In such more common scenarios, it is not required that the user defines the method `run_and_wait_end` (see line 16 in Code 14) or `run` for each single node. The pipeline building block already provides these methods.

7.2.2 Multi-input and multi-output *nodes*

The `ff_minode_t` and the `ff_monode_t` classes define a *multi-input* and a *multi-output* FastFlow node, respectively. Both classes extend the `ff_node_t` class. Differently from the basic *ff_node_t*, which has one input channel and one output channel, the multi-input node may have many input channels (at least one) while the multi-output node may have many output channels (at least one). The number of input/output channels is not defined when creating the object, instead they are associated to the multi node when it is connected to other nodes. In Figure 7-1 is shown how a multi-input and a multi-output nodes are connected to standard nodes.

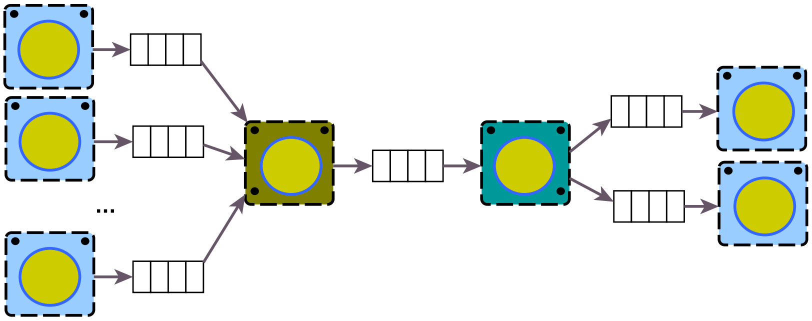


Figure 7-1: Channels associated to *multi-input/output* nodes.

For the multi-input node, as soon as a data element is received from one of its input channels, the `svc` method is called by the FastFlow RTS. Its default gathering policy is to collect data elements "from any input channels" in a non-deterministic fashion. The node terminates when it receives the special value `EOS` from *all* input channels, and then the `EOS` value is propagated into the output channel (if present). The *multi-input* node also provides the method `all_gather` that allows the user to synchronously receive a data element from all input channels. This method may be called inside the svc method once a single input item has already been received to complete the reception of the other items from other input channels. This method is particularly useful to implement the *Map* pattern.

Concerning the multi-output node, it has one single input channel and many output channels. What characterizes the *multi-output* node is that it offers the possibility to control to which output channel the data elements computed in the svc method have to be forwarded. It allows having full control of data routing by using the method ff_send_out_to. Differently from the ff_send_out function, the ff_send_out_to method permits to specify the output channel identifier (from 0 to $n-1$) where the data has to be delivered. The default forwarding method for the elements returned from the svc method is the *round-robin* policy. Alternatively, the user can set the so-called *on-demand* policy which allows distributing data elements in a way that produces a balanced workload distribution among the output channels. This policy is described with more detail in Section 7.3.2.

7.2.3 Sequential nodes combiner

Sequential FastFlow nodes can be combined by using the *combiner* building block implemented by the class ff_comb. Given two sequential building blocks that potentially could be executed in pipeline, combining them is an operation that statically merges the two nodes in a single concurrent entity. Conceptually, the operation of combining sequential nodes is similar to the functional composition of two functions. In this case the functions that will be composed are the service functions of the two nodes, i.e. svc, svc_init and svc_end methods as well as the eosnotify method. This building block will be used to introduce automatic transformation of the application concurrency graph (see Chaptert 8).

The ff_comb class, takes two sequential nodes as input arguments and returns a new FastFlow node capable to execute the business logic of both input nodes. This operation allows us to not modify the original code of the two objects implementing the nodes that are combined. Together with the C++ class ff_comb, the building block layer also provides the helper function combine_nodes that gets two FastFlow

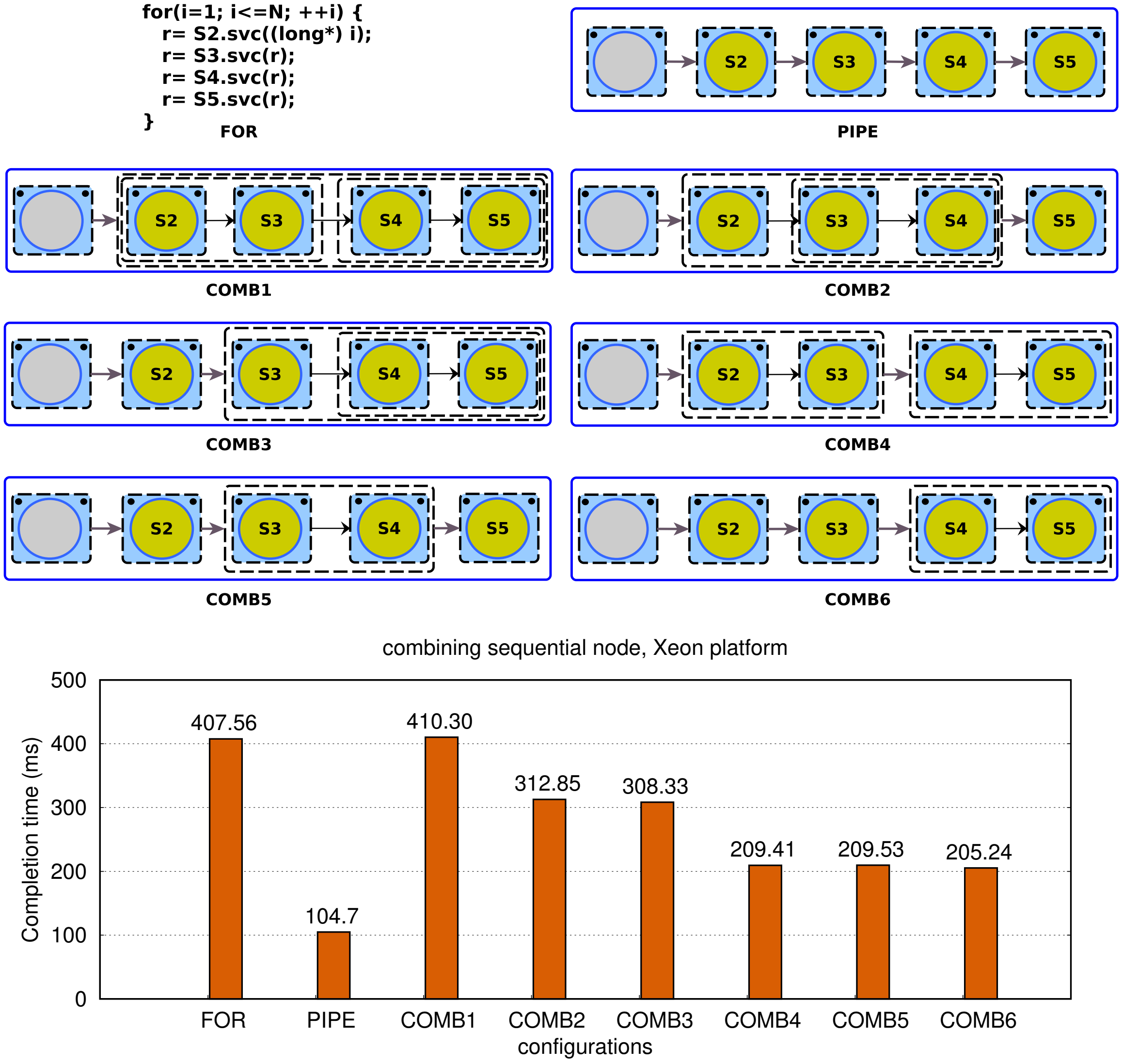
```cpp
#include <ff/ff.hpp>
using namespace ff;

struct First: ff_node_t<long> {
  int svc_init() {
    V.resize(1000);
    std::for_each(V.begin(),V.end(), [&](long& i) {i=1;});
    return 0;
  }
  long *svc(long*) {
    for(size_t i=0;i<V.size();++i) ff_send_out(&V[i]);
    return EOS;
  }
  std::vector<long> V;
} _1;

struct Inc: ff_minode_t<long> { // multi-input node
  long *svc(long *in) { auto& t(*in); ++t; return in;  }
} _2;

struct Dec: ff_monode_t<long> { // multi-output node
  long *svc(long *in) { auto& t(*in); --t;  return in; }
} _3;

struct Last: ff_node_t<long> {
  long *svc(long *t) { std::cout<<*t<<"\n"; return GO_ON;}
} _4;

int main() {
  auto comb= combine_nodes(_1,
                   combine_nodes(_2,combine_nodes(_3, _4)));
  comb.run();  // asynchronous execution
  // ... other code here
  comb.wait(); // wait for node termination
  return 0;
}
```

Code 15: Combining multiple FastFlow nodes by using the `combine_nodes` function.

nodes and returns a `const ff_comb` object. This helper function simplifies the usage of the combiner building block allowing also to concatenate multiple combine operations to obtain a single node entity. The Code 15 shows a simple example that combines several FastFlow nodes by using the `combine_nodes` helper function.

7.2.4 Evaluation

In this section we evaluate the *sequential nodes combiner* building block implemented by the FastFlow class *ff_comb*. The experiments were conducted on the *Xeon* platform (see Table 6.1).



```
for(i=1; i<=N; ++i) {
   r= S2.svc((long*) i);
   r= S3.svc(r);
   r= S4.svc(r);
   r= S5.svc(r);
}
```

Figure 7-2: Different cases tested for the *sequential node combiner* building block (`ff_comb`). Each stage of the pipeline S_X, $X \in [2..5]$ has a service time of about 1us.

The test performed is straightforward and aims at assessing the overhead introduced by the *sequential combiner* building block. We considered a five-stage pipeline of standard sequential nodes. The first stage just produces a stream of N values and then terminates. This stage is kept as an independent pipeline stage and never combined to other stages. The other four stages, each one spends about $1us$ of local computation upon receiving an input value and then forwards the value to the next stage (if any). We consider different configurations by combining pipeline stages according to the schemas showed in Figure 7-2. The base case is the one where the entire computation is executed in a sequential loop and invoking the service functions of each node composing the logical pipeline (the case with the label "FOR" in Figure 7-2). At the other extreme, another interesting case to compare with is when the entire computation is executed without combining any stage, i.e. executing all stages in pipeline (this is the case with the label "PIPE" in Figure 7-2). The expected completion time (T_c) for the pipeline execution (PIPE) of this simple test is:

$$T_c \cong N \cdot \max_{X \in [1..5]} \{T_{S_X}\}$$

where T_{S_X} is the service time of stage S_X. In our test the service time for each stage is about $1\mu s$ and the number of data elements flowing is $N = 100,000$, therefore the ideal completion time is $T_c \cong$100ms.

The results of the test are reported in Figure 7-2. As can be seen, the obtained results for each test case are very close to the expected value. For example, for the cases "COMB2" and "COMB3" the ideal time is 300ms whereas for the cases "COMB4", "COMB5" and "COMB6" the ideal time is about 200ms. The results obtained differ for less than 5% from the theoretical optimal value.

The second test we performed aims to study the overhead introduced by the sequential combiner when an increasing number of sequential nodes are combined. In this case a three-stage pipeline is considered. The first and last stage are fixed and never combined to other stages. The middle stages (minimum two) are all combined in a single node using the `ff_comb` building blocks. As in the previous test, the average service time of the stage is about 1us (the schema is the one sketched at the top of Figure 7-3). The results obtained are shown in Figure 7-3 where the number of middle stages vary from $M = 2$ to $M = 512$. In the plot are reported the overhead per data

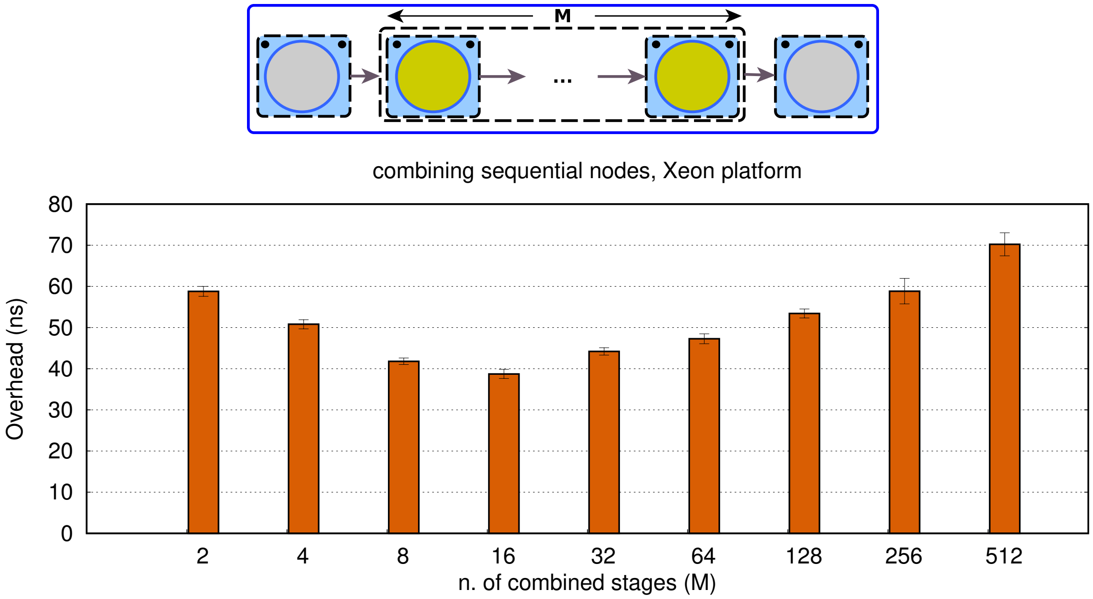


Figure 7-3: Per task overhead introduced by the *sequential node combiner* building block varying the number of nodes combined (M).

element ($N = 100,000$) in nanoseconds. As can be seen, the overhead introduced is small and slightly increases with the number of nodes combined starting from 16 nodes.

In conclusion, the tests performed showed that the *sequential nodes combiner* does not introduce significant overhead when merging sequential nodes. Its low overhead enables the possibility to use the *nodes combiner* as a mean to reduce the number of concurrent entities in the FastFlow network and to introduce static transformations/optimizations of the concurrency graph describing the FastFlow application (see Chapter 8).

7.3 Parallel Building Blocks

In this section we introduce the parallel building blocks used for connecting and coordinating both standard FastFlow nodes as well as multi nodes. First we show how to connect nodes by using the *pipeline* building block. Despite the name, in the context of FastFlow building blocks the pipeline component does not produce just linear chains of nodes, instead it allows us to connect sequential and parallel building blocks creating non-linear cyclic directed graphs. The *pipeline* building block enables data-flow pipeline execution of its nodes (also called stages). The other two building blocks presented in this section are the *farm* and the *all-to-all*.

7.3.1 Connecting FastFlow nodes in pipeline

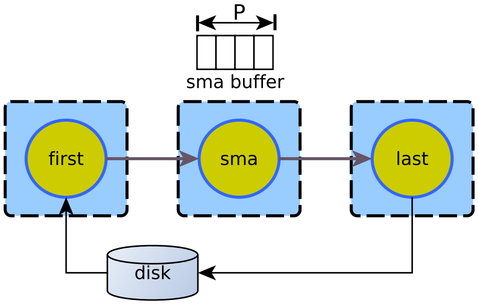
Connecting nodes implies connecting the output channels of a node with input channels of one or more other nodes. The building block that allows connecting nodes is the *pipeline* implemented by the FastFlow class `ff_pipelipe`. The `ff_pipeline` itself is a FastFlow node, so multiple networks of FastFlow nodes built independently, can then be connected together by adding them to a *pipeline* building block, thus creating more complex data-flow topologies. To increase type safety, the template class `ff_Pipe`, which extends the `ff_pipeline` class, can also be used. It statically checks type matching between two subsequent nodes of the pipeline. The `svc` method of the first node of the outermost pipeline, that usually does not have any input channels (it could have at least one if a feedback channel was present in the pipeline), is invoked by the RTS passing as input parameter a `nullptr` pointer.

As an example of pipeline, let us consider the problem of computing a *Simple Moving Average* (SMA) that is a method for computing an average of a stream of numbers by only averaging the last P number from the input stream, where P is known as the period. The input numbers are contained in a file and the results will be stored in a file. The user may define three FastFlow nodes: the first one reads numbers from the input file and produces a stream of them; the second one computes the SMA, and finally the third stage stores the results into the output file. Code 16 shows how to build and (synchronously) execute the three-stage pipeline.

Creating cyclic network of nodes

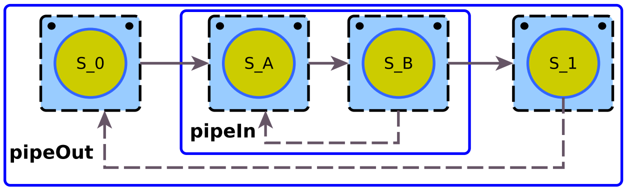
The backward channel between two nodes is called *feedback channel*. These channels are used either to route back results to some previous node or to send notification messages that can be useful to make decisions, for example, on how to route next

```
#include <ff/ff.hpp>
First first("in.txt");
Second sma(P);
Third  third("out.txt");
ff_Pipe<> pipe(first,sma,third);
if (pipe.run_and_wait_end()<0)
  error("running pipe");
```



Code 16: A simple three stage linear pipeline.

```
1 #include <ff/ff.hpp>
2 using namespace ff;
3 ff_Pipe<> pipeIn(S_A,S_B);
4 pipeIn.wrap_around();
5 ff_Pipe<> pipeOut(S_0,pipeIn,S_1);
6 pipeOut.wrap_around();
7 if (pipeOut.run_and_wait_end()<0)
8   error("running pipe");
```



Code 17: Creating nested pipelines with feedback channels.

data elements or about how to regulate the injection rate. As we will see when presenting the `ff_farm` building block, by leveraging feedback channels it is possible to implement dynamic data scheduling policies between the Emitter node, which is a multi-output node, and a pool of Worker nodes.

Feedback channels may be introduced only in parallel components, (i.e. `ff_a2a`, `ff_farm` and `ff_pipeline`) by using the method `wrap_around`. The feedback channel for a building block, acts as a *modifier* of its input or output channel set, enabling the creation of one or more extra channels going in the opposite direction with respect to the standard pipeline data flow.

In Code 17 it is defined a pipeline (`pipeIn` – line 3) composed by two stages (`S_A` and `S_B`) connected by a forward and a feedback channel (the last one created at line 4). The `pipeIn` pipeline is then added as a middle stage of a three-stage pipeline called `pipeOut` (line 5). The `pipeOut` pipeline has a feedback channel created by using the method `wrap_around` (line 6). S_A is a *multi-input* node while S_B is a *multi-*

output node. To discern between which output channel to select in the S_B node, the FastFlow RTS defines the nomenclature for input/output channels as described in Figure 7-4.

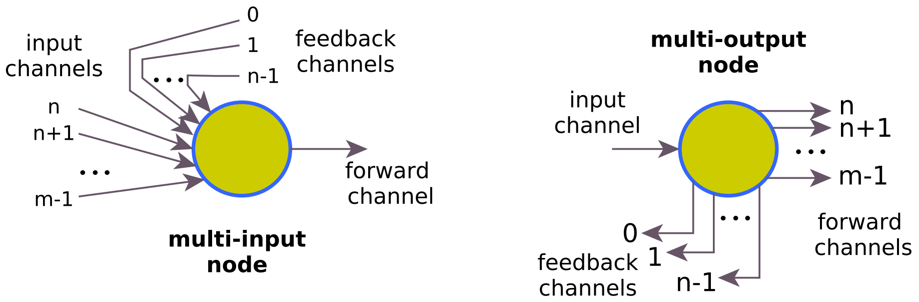


Figure 7-4: Nomenclature of input/output channels in *multi-input* and *multi-output* nodes.

The number of input and output channels, currently active for a given multi node, can be obtained through specific member functions of the respective objects. Specifically, for the multi-input node it is possible to know the number of forward channels and the number of feedback channels by using the methods `get_num_outchannels()` and `get_num_feedbackchannels()`, respectively. For the multi-input node the number of input channels can be obtained through `get_num_inchannels()` whereas the id of the channel where the current data element has been received can be obtained by the method `get_channel_id()`. To know if the input element is coming from an input or a feedback channel the library provides the method `fromInput()` which returns *true* in the first case and *false* in the second case.

Avoiding deadlock caused by bounded buffers

One of the most common causes of deadlock in data-flow networks including cycles happens when some nodes cannot make progress because of the bounded capacity of channels. Channels with bounded buffers can fill up quickly during traffic burst, and if the application topology graph has cycles, deadlock can occur quickly as well.

To avoid potential deadlock induced by the presence of cycles produced by feedback channels, every queue associated with a backward channel is instantiated with an unbounded capacity by the FastFlow RTS. Therefore a send operation on a feedback channel never blocks the sender node.

Moreover, multi-input nodes having both input channels coming from previous stages and feedback channels coming from some following stage, the RTS prioritizes the management of data items coming from feedback channels for draining the internal networks of the parallel building block.

These simple strategies, together with a careful management of feedback channels at application or pattern level, are typically enough to avoid deadlock situations caused by a limited capacity of buffers in the network topology composed by the concurrent activities generated by the FastFlow application.

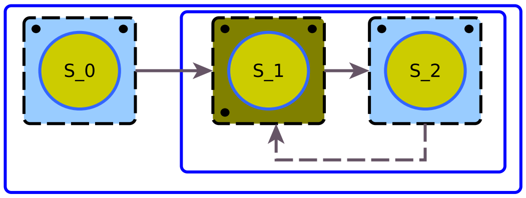
Managing program termination

If the application graph contains one or more cycles, one critical aspect is the management of program termination. As discussed in Sec. 7.2.2, a FastFlow node terminates if and only if it receives the `EOS` value from *all* input channels. Then, the `EOS` value is propagated into all output channels. Therefore, the `EOS` generated by the first stage of a pipeline will be automatically propagated over the entire network to terminate all nodes.

If a node has in input both a forward channel coming from a previous stage and a feedback channel coming from one of the following stages in the pipeline, the `EOS` value received from the previous stage in the pipeline will not be automatically propagated until the `EOS` is received from the feedback channel too. This situation, that prevents program termination, can be handled by using the *eosnotify* method that is called by the RTS as soon as an `EOS` value has been received in one of the input channels. The method provides as argument the channel identifier where the `EOS` value has been received. By counting the number of *on-the-fly* data elements that have not come back yet through the feedback channels, and considering the channel-id from which the `EOS` message has been received, it is possible to decide when the termination message can be propagated to the next stage.

Code 18 shows a generic *multi-input* node that has in input both a standard input

```
 1 template<typename T>
 2 struct S_1:ff_minode_t<T>{
 3   T* svc(T* in) {
 4     if (!this->fromInput()) {
 5       --on_the_fly;
 6       if ((on_the_fly==0)&&eosreceived)
 7         return this->EOS;
 8       return this->GO_ON;
 9     }
10     on_the_fly++;
11     return in;
12   }
13   void eosnotify(ssize_t id) {
14     if (!eosreceived) {
15       eosreceived = true;
16       if (on_the_fly==0)
17         this->ff_send_out(this->EOS);
18     }
19   }
20   bool eosreceived=false;
21   long on_the_fly=0;
22 };
```



Code 18: Termination management using the `eosnotify` method in *multi-input* nodes with *feedback* channels in input.

channel and a feedback channel (it is the code executed by the stage labeled with *S_1* in the right-hand side). In line 4 we check if the input message just received comes from the standard input channel or from feedback channels. If the message comes from a feedback channel, then we decrease the number of on-the-fly messages (i.e. messages we sent to the next stage and that have not come back yet) and in the case of the `EOS` message has already been received (line 15) and the number of on-the-fly messages is zero, then the `EOS` message can be forwarded to the next stage (line 7). The other case in which the `EOS` has to be forwarded is when we receive the termination message, and all previous messages have already come back (line 17).

It is worth mentioning that the *eosnotify* method can also be used to flush any internal buffer of the node once the `EOS` message has been received and before the `svc_end` method is called by the RTS.

7.3.2 The ff_farm building block

The FastFlow *farm* building block is a flexible implementation of the *Task-Farm* pattern (see Section 2.5.3). Basically it models functional replication coordinated by a master node. However, it can be used also to compute in parallel n distinct functions on the same input data, modeling the execution of "Multiple Instructions Single Data" (MISD).

The simplest form of the *farm* building block (implemented by the C++ class `ff_farm`) is composed by two concurrent entities executed in pipeline: a multi-output node called *Emitter* (the master), and a pool of standard nodes called *Workers*, each of them possibly executing the same business logic code. The *Emitter* node schedules the data elements received in input toward the *Workers* using either a default policy (i.e. *round-robin* or *on-demand*) or according to the algorithm implemented by the user code defined in its `svc` method. In this second scenario, the data routing is controlled by using both the `ff_send_out_to` as well as the `broadcast_task` methods of the multi-input node implementing the *Emitter*.

The `ff_farm` extends the basic `ff_node` building block and so it is itself a FastFlow node object. The pool of *Workers* may be any object extending the `ff_node` building block, therefore the single Worker can be implemented by any instance of `ff_pipeline`. In other words, `ff_farm` and `ff_pipeline` objects can be composed and nested in any possible combinations. For simplicity sake, the farm's Worker can also be an `ff_farm` as well as an *all-to-all* building block implemented by the class `ff_a2a`. The *Emitter* entity of a *farm* building block can be either a user-defined standard node or a multi node.

Since the *farm* building block is a primitive building block it has additional features compared to the mere composition of a multi-output node and a set of FastFlow nodes. Specifically, the Emitter entity is able to receive from multiple input channels, for example it can receive from all *Workers* of a previous *farm* in pipeline, therefore it acts as a *multi-input* and *multi-output* node at the same time. Another important feature of the *farm* building block is the possibility to keep the input ordering of data element and to throttle the number of `Workers` dynamically (we discuss both these important features in dedicated sections below).

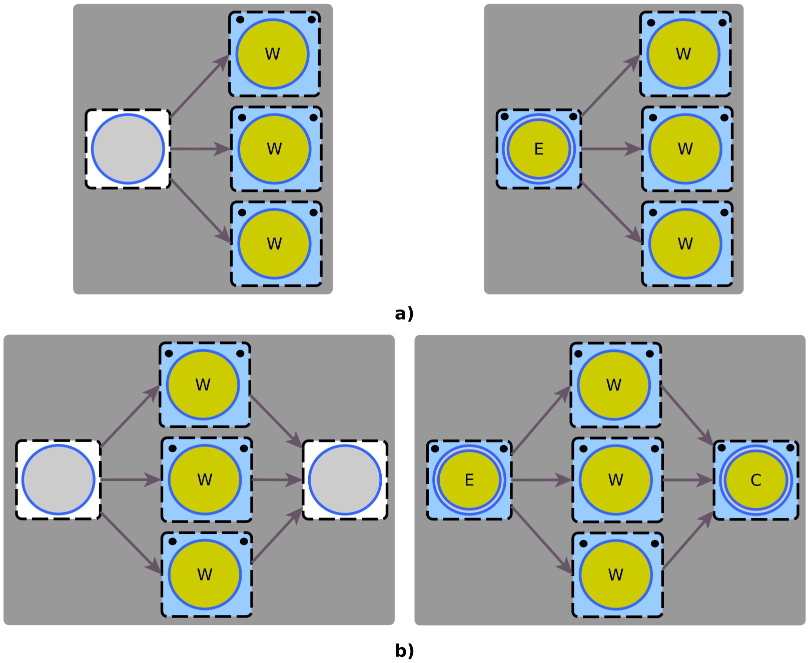


Figure 7-5: Different topologies of the `ff_farm` building block using standard FastFlow nodes: a) the default farm building block; b) farm with the Collector node.

The two base parallel schema of the *farm* building block are sketched in Figure 7-5 (for simplicity, in the figure are used only standard sequential nodes). In the case of a farm configured to preserve data ordering, it must have a Collector entity which gathers data elements being computed by the pool of *Workers*. The Collector may be either a user-defined standard node or a multi-input node. The Collector node can be added to the farm building block whenever a particular data gathering policy is needed (for example for executing an `all_gather` – see Sect 7.2.2) and/or when a post-processing phase has to be applied to the data elements produced by the *Workers* before they leave the farm building block. The user may define the *farm* Collector as a standard FastFlow node as well as a *multi-input/output* node. In the latter case, the Collector acts as a *multi-input* and *multi-output* node at the same time. From the concurrent execution point of view, the three farm entities (Emitter, Workers, and Collector) work according to a data-flow pipeline execution model over a stream of input elements.

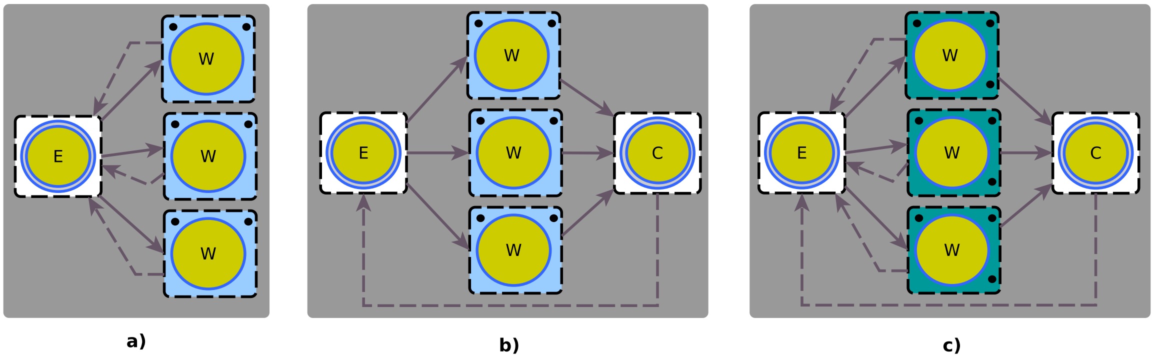


Figure 7-6: Possible topologies of a farm with feedback channels: a) each Worker has a channel toward the Emitter; b) the Collector has a feedback channel toward the Emitter; c) merging of topology a) and b).

The *feedback modifier* applied to the `ff_farm` building block allows the programmer to produce different network topologies. All possible *farm* topologies are sketched in Figure 7-6.

There are several ways to construct a farm building block object. The simplest one is to create a farm without defining both Emitter and Collector entities and using the default implementations provided by the building block itself. This simple case is reported in the Code 19 (left-hand side). A slightly more elaborate scenario is sketched at the right-hand side of Code 19 where the farm building block is built providing a user-defined Emitter and Collector nodes. The scheduling policy of input data elements is set to *on-demand* (line 12) and the *Worker* resources allocated at line 10 will be automatically released once the farm object is destroyed (line 13). The methods, `cleanup_emitter`, `cleanup_collector`, `cleanup_workers` and `cleanup_all` allow the programmer to delegate the destruction of the objects to the *farm*'s destructor.

Scheduling input elements. To select the *Worker* where an incoming input data element has to be sent, the FastFlow `farm` building block uses an internal object called `ff_loadbalancer`. By overriding specific methods of the load-balancer class, it is possible to define new scheduling strategies for the farm Emitter. Currently, two policies have been implemented and provided to the user:

```
#include <ff/ff.hpp>
using namespace ff;
// user-defined worker
Worker    W;
// creating the pool of workers
std::vector<ff_node*> V;
for(int i=0;i<nworkers;++i)
  V.push_back(new Worker(W));
// a farm with default emitter
// and collector
ff_farm farm(V);
if (farm.run_and_wait_end()<0)
  error("running farm");
```

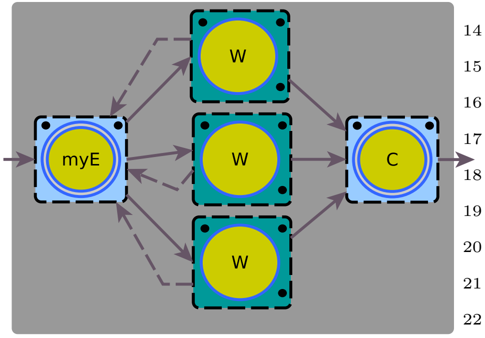
```
#include <ff/ff.hpp>
using namespace ff;
// user-defined ...
Emitter   E; // ... emitter
Collector C; // ... collector
Worker    W;
// creating the pool of workers
std::vector<ff_node*> V;
for(int i=0;i<nworkers;++i)
  V.push_back(new Worker(W));
ff_farm farm(V,E,C);
farm.set_scheduling_ondemand();
farm.cleanup_workers();
if (farm.run_and_wait_end()<0)
  error("running farm");
```

Code 19: Creating a farm with default emitter and collector (left), and with user-defined emitter and collector (right).

- *loose round-robin.* This is the default farm scheduling policy. The Emitter sends data elements in a round-robin fashion to the Workers in the pool. If the input queue of the Worker selected is full (when the channels have bounded capacity), the Emitter does not wait until it can insert the element into that queue; instead, it selects the next Worker and keeps going on until the element can be assigned to one of the Workers. It is clear that this policy follows a round-robin selection of Workers only if the queues have an unbounded capacity (that is why we have called this policy “loose” round-robin). In the remaining, we will refer to this policy simply as *round-robin.*

- *on-demand.* This is a simple implementation of the “auto-scheduling” policy. The semantics of this policy is that the Worker “ask for” a new data element rather than passively accepting elements sent by the Emitter. Such distribution policy can be employed by calling the method `set_scheduling_ondemand()` on the farm object. By default, the asynchrony level of the “request-reply” protocol between the Emitter and the generic Worker is set to one. If needed, it can be increased by passing an integer value greater than zero to the farm’s method

```
1 ff_farm farm;
2 MyEmitter myE;
3 farm.add_workers(Workers);
4 farm.add_emitter(myE);
5 farm.wrap_around();
6 farm.add_collector(C);
7 ....
```



```
1  template<typename T>
2  struct MyEmitter: ff_monode_t<T> {
3   T* svc(T* in) {
4     int wid = get_channel_id();
5     if (wid == -1) {
6       int victim = selectReadyWorker();
7       if (victim<0) data.push_back(in);
8       else
9         ff_send_out_to(in, victim);
10      return GO_ON;
11    }
12    if (data.size()>0) {
13      ff_send_out_to(data.back(), wid);
14      data.pop_back();
15    } else
16      updateReadyWorkers(wid);
17    if (checkTermination()) return EOS;
18    return GO_ON;
19  }
20  void eosnotify(ssize_t chid) { ... };
21  std::vector<T*> data;
22 };
```

Code 20: Example showing how to define an Emitter node with a custom scheduling policy.

`set_scheduling_ondemand()`.

The round-robin policy is very simple to implement and consequently very efficient. However, it does not work particularly well if input data elements have very different execution times. On the other hand, the on-demand scheduling policy has a slightly higher overhead than the round-robin policy but can ensure almost even workload distribution among Workers. The user can implement an application-specific scheduling policy by implementing a customized farm Emitter leveraging the `ff_loadbalancer` object of the farm. By utilizing the the method `ff_send_out_to`, each input data element can be sent to a specific destination Worker.

A snippet of code showing how to implement a simple user-defined policy in the Emitter node is shown in Code 20. In line 2 at the top-left of Code 20 is shown how to created an instance of the MyEmitter node class. The Emitter receives messages both from the previous stage as well as from the Workers. If the message is coming from the previous stage (line 5) the `selectReadyWorker` looks for a Worker that is waiting to receive a message (i.e. a Worker that has already completed its previous job and has already sent a notification message to the Emitter) and if the search succeeds the input data element is sent to the selected Worker (line 9) otherwise the input message is stored into a local buffer (line 7). If the input message comes from one of the Workers and if there are elements in the local buffer, then the oldest data element received is sent to the current Worker (line 13) otherwise the Worker is considered "ready" (line 16). The termination condition is checked before exiting the service function (line 17). The condition is that the `EOS` message has been already received (therefore the `eosnotify` function has been called by the FastFlow RTS) and there are no more elements in the local buffer.

Throttling the number of Workers.

Concurrency throttling refers to the possibility to dynamically change the number of threads composing a parallel application [114]. It is a powerful mechanism through which it is possible to increase or decrease the concurrency level of an application to improve performance and/or reduce power consumption [131]. Instead of placing many threads on a smaller set of cores (this technique is called *Thread packing* [92]), the concurrency manager is responsible for reducing their number to decrease contention on shared resources. Concurrency throttling is a technique that cannot be used in some cases since it may require some interactions with the application itself and with the RTS. For example, it may involve a redistribution of thread's internal state, an operation that usually requires a synchronization protocol to keep the state consistent. This is a complex problem to deal with in the general case and can be easier to attack if the parallel structure of the application is known in advance (e.g., a farm skeleton) and the kind of internal state used are known [130].

The FastFlow library offers the mechanisms to change the number of Workers in a farm on-the-fly without stopping and restarting the entire application. This feature, enables the possibility to build parallel patterns, and more generally, structured applications modeled as composition of parallel patterns that dynamically adapt the number of resources employed according to, for example, the input rate [126] or to performance/power constraints [131].

In the farm building block, the number of Workers can be greater than the number of resources available. All Workers, or a subset of them, can be dynamically terminated (or temporarily stopped) by selectively sending to them a particular message, `EOSW`. This message is not propagated by the RTS outside the farm building block. The farm Emitter enqueues the `EOSW` message into the input channel of the selected nodes corresponding to the Workers that have to be terminated to start their termination protocol. All messages already present in the queues of the selected Workers will be processed before the Workers' termination. In the FastFlow library threads implementing Worker nodes are not created and destroyed dynamically, instead they are temporally stopped and started again when needed.

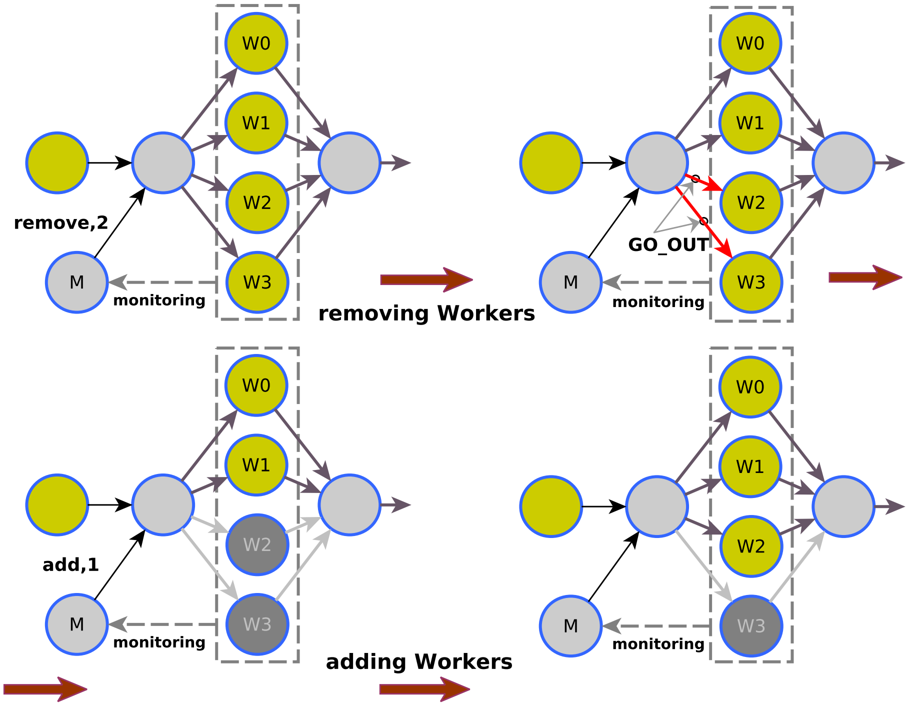


Figure 7-7: Logical schema for removing and adding Workers in a farm.

In addition, it is possible to put to sleep and afterward wake up one or more sleeping Workers upon request. To enable this feature, the generic Worker has to have the flag `freezing` set to `true`. This is an internal flag of the FastFlow sequential nodes that tells the RTS to put the thread running the node to sleep instead of terminating it when it receives an EOSW message. This can be done is several ways, as for example by starting the farm building block by using the method run_then_freeze (instead of run_and_wait_end) or directly by calling the freeze() method on the farm object. If a Worker is started with the "freezing flag" enabled, once it receives in its input queue an EOSW message or a GO_OUT message, then the RTS does not terminate the threads, instead, the Worker will be suspended on an event object. The difference between the EOSW and GO_OUT messages is that when the node receives the EOSW message the eosnotify callback is called and the message is propagated until it reaches the farm's Collector. This is not the case for the GO_OUT message. The svc_end methods will be called in either case to notify the user's code that the node is going to be suspended.

The threads running the node will be woken up at the next building block execution. This simple protocol allows us, for example, to spare the overhead associated to thread creation if the building block will be activated many times during the application execution. However, to dynamically add and remove Workers within the same application run, the method thaw(worker-id) allows the programmer to restart anytime the thread running the Worker node with the logical identifier worker-id. The restarted Workers will continue executing the main loop and processing data elements in their input queues. The typical configuration of a FastFlow application exploiting dynamic concurrency throttling in the task-farm pattern is sketched Figure 7-7. A manager node (M in the figure) is attached to the farm Emitter by a point-to-point channel. The same manager node can be attached to multiple farms building blocks. The commands sent by the manager to add and remove Workers will be received by the Emitter node. If the message comes from the manager node (such channel has a unique identifier), then instead of forwarding the message to one of the Workers it is interpreted and the corresponding action executed by the Emitter node. In Figure 7-7, it is shown a sequence of executions of two commands: the first command asks to remove two Workers, while the second one asks to add one Worker (it is not possible to ask for more Workers than the ones currently stopped). For the remove command, the Emitter will send the EOSW message to the two selected Workers (the

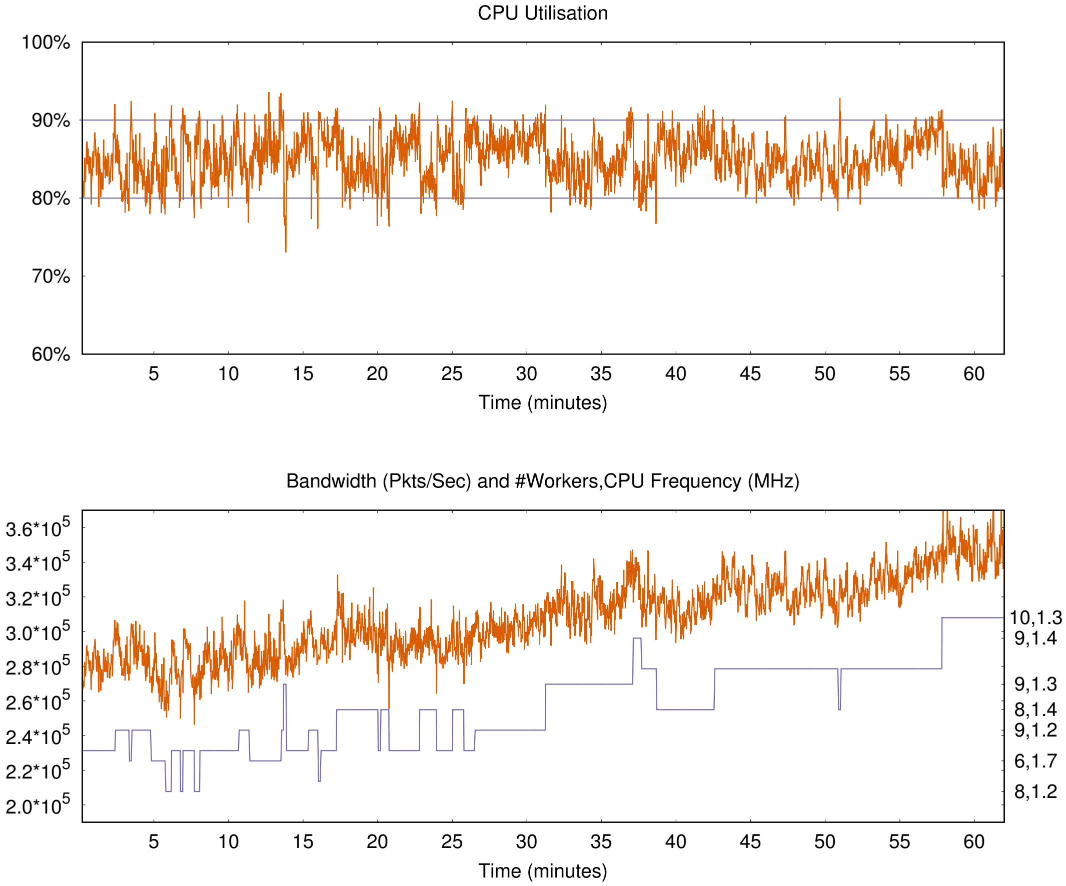


Figure 7-8: FastFlow concurrency throttling used in a DPI application. Top plot: CPU utilization within the time interval considered. Bottom plot: comparison between sustained bandwidth and system configuration (n. of Workers, CPU-frequency).

Workers that are removed are those with the highest id so to keep the list of identifiers continuous). By reading monitoring information collected by each node, the manager node knows when the Workers will be stopped. The second command is a request to add one Worker therefore the Emitter will execute the `thaw` method on the object corresponding to the stopped Workers with the lowest id. After the `thaw` method completes, the Emitter restarts sending data-elements also to that Worker.

The manager uses monitoring information collected by Workers to make decisions. An example of how the FastFlow concurrency throttling mechanisms have been used for a real network application is shown in Figure 7-8. The step function in the figure represents the product between the number of active Workers and the operating frequency of the CPUs. For example, at minute 30 nine farm's Workers are used with a system clock of 1.2Ghz, while at minute 35 eight Workers are used with a clock frequency of 1.4Ghz.

The application analyses one hour of HTTP packets flowing in a backbone network, searching for well-known patterns identifying possible security threats. It has been implemented by using the Peafowl framework [110], a flexible and extensible Deep Packet Inspection (DPI) framework implemented on top of the FastFlow library. The logical structure of the application is a single farm where the Emitter reads networks packets, the Workers identify the protocol and process the data content, and the Collector either drops the packets or re-injects them into the network.

In this application, the manager node changes both the number of Workers of the farm and also the CPUs frequency of the system with the primary objective to keep the utilization of the system high (in the test reported here, always between 80% and 90%, see the top part of Figure 7-8) while minimizing system power consumption.

To do this, among all possible configurations given by the number of farm's Workers and CPU frequency that are able to sustain the input rate, the manager selects the one with the lower power consumption according to a performance model. The details of the model used to select the optimal configuration is presented in Danelutto et al. [115]. The number of Workers is changed sending commands to the Emitter node according to the protocol described above, whereas CPU frequency is changed directly by the manager by using the `cpufreq` library.

The mechanisms and protocols described in this section, have recently been used in the Nornir framework [131, 293] to enforce performance and power consumption constraints, and also to study different techniques to efficiently handle out-of-order and bursty data streams [246].

Finally, it is worth to point out that all FastFlow nodes can be suspended and re-started by properly invoking the methods `freeze` and `thaw`. Therefore, though not yet implemented, the *concurrency throttling* mechanism can also be provided for the *all-to-all* building block.

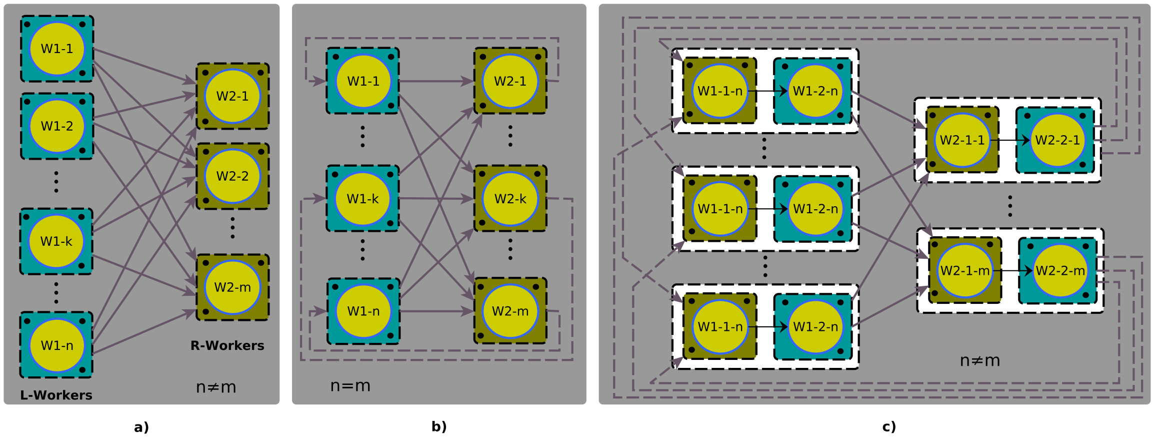


Figure 7-9: All-to-all (A2A) building block topologies: a) default schema; b) A2A with feedback channels, the two sets have the same cardinality; c) A2A with feedback channels, the two sets have different cardinalities.

7.3.3 The ff_a2a building block

The farm building block is a very powerful and flexible component with its many features and possible configurations. However, in some applications, the Emitter and/or the Collector nodes may introduce a centralized point of control that could become the bottleneck of the entire processes network, hence preventing scalability. In these cases, the *all-to-all* building block (briefly A2A) can be used to avoid any centralization point. It defines two distinct sets of *Workers* connected together such that a Worker of the first set (called L-Worker set, or simply L-Workers) is connected to all Workers of the second set (called R-Workers). If the number of L-Workers is m and the number of R-Workers is n, then there are $m \cdot n$ communication channels connecting the two sets. This communication pattern is also known as *shuffle*. From the concurrency standpoint, the semantics of the *all-to-all* building block is that of two *farm*s connected in pipeline, the first running the L-Workers and the second running the R-Workers, with no guarantee on the arrival ordering of data elements to the R-Workers.

The FastFlow class `ff_a2a` provides the interface for defining an instance of the A2A building block. The possible logical schemas of the A2A building block are sketched in Figure 7-9.

The L-Workers are multi-output nodes so each one is able to select the destination

```cpp
#include <ff/ff.hpp>
using namespace ff;
// user-defined workers
Worker1    W1;
Worker2    W2;

// creating L-Workers
std::vector<ff_node*> V1;
for(int i=0;i<nworkers1;++i)
  V1.push_back(new Worker1(W1));

// creating R-Workers
std::vector<ff_node*> V2;
for(int i=0;i<nworkers2;++i)
  V2.push_back(new Worker2(W2));

ff_a2a a2a;
// adding the first set and setting
// the on-demand policy with
// asynchrony degree 2
a2a.add_firstset(V1, 2);
a2a.add_secondset(V2);
if (a2a.run_and_wait_end()<0)
  error("running a2a");
```

```cpp
#include <ff/ff.hpp>
using namespace ff;
// user-defined workers
Worker1    W1; // standard node
Worker2    W2; // multi-output node
MultiInputHelper1 H1;// helper node
MultiInputHelper2 H2;// helper node

// creating the L-Workers
std::vector<ff_node*> V1;
for(int i=0;i<nworkers1;++i)
  V1.push_back(new ff_comb(H1,W1));

// creating the R-Workers
std::vector<ff_node*> V2;
for(int i=0;i<nworkers2;++i)
  V2.push_back(new ff_comb(H2,W2));

ff_a2a a2a;
a2a.add_firstset(V1, 0, true);
a2a.add_secondset(V2, true);
a2a.wrap_around();
if (a2a.run_and_wait_end()<0)
  error("running a2a");
```

Code 21: Two examples showing how to define an *all-to-all* building block.

Worker of the second set. As for the farm, the default scheduling policy of data elements is the round-robin one. The on-demand policy can also be used with a user-defined asynchrony degree. The user may also have full control of the destination of output messages by using the `ff_send_out_to` method.

The R-Workers are multi-input nodes, so each one receives data elements from any L-Workers. Since R-Workers are multi-input nodes, the `all_gather` method can also be used to implement a *gather-all* collecting policy.

The A2A extends the standard ff_node so it can be used as a farm's worker or as a stage of a pipeline. Allowed compositions of A2A with other building blocks are described in Chapter 5.

Code 21 shows two basic examples on how to define and execute an A2A building block. In the left-hand side of the figure, it is created an A2A where the L-Workers use an on-demand scheduling policy to distribute data elements to R-Workers. This is enabled in line 21 where an asynchrony level of two messages is configured. If the second parameter of the method `add_firstset` is zero or it is not set, then the default round-robin policy will be used. In the right-hand side of Figure 21, an A2A with feedback channels is built. In this case we suppose that `Worker1` is a standard node while `Worker2` is a multi-output node. Since the number of Workers in the first set and in the second set might be different, two interface nodes are needed to adjust the cardinality of the two sets (see also Section 5.3). These two interface nodes act as "helper nodes" to set the proper cardinality. They do not execute any business logic. Specifically, a *sequential combiner node* building block is used to create two new nodes that are added in the L-Worker set and in the R-Worker set. They are created at line 12 and at line 17, respectively. Both nodes `H1` and `H2` (defined at line 6 and line 7, respectively) are *multi-input* nodes. This way, all nodes in the L-Worker set are *multi-input* nodes while all nodes in the R-Worker set are *multi-output* nodes, hence they can be connected with feedback channels regardless of their number. As we shall see in Chapter 8, helper nodes can be added automatically by the FastFlow RTS when the concurrency graph composed by all nodes describing the application is transformed to optimize the number of concurrent nodes.

Main differences between *farm* and *all-to-all.* If the A2A building block is configured to have one single L-Worker and many R-Workers, the resulting parallel structure is semantically equivalent to that of a *farm* building block without the Collector. In principle, it is possible to implement a *farm* with an A2A, being a *farm* (without Collector) a particular instance of the A2A.

Even though we could have provided the user with only the *all-to-all* building block without adding the *farm* with no Collector in the building block set, we decided to follow a different path. The *farm* is easier to use than the A2A. It is a more specialized component with *ready-to-use* features that the user can simply enable without extra coding. For example, a *farm* can preserve input data ordering and the Emitter is *de facto* both a *multi-input* and a *multi-output* node allowing to connect in pipeline multiple farms without the Collector. The *multi-output* L-Workers nodes cannot receive from a multitude of input channels. To solve this issue, the user has to create a combiner node for each L-Worker, where each combiner node combines a *multi-input* and a *multi-output* node. In addition, while for the *farm* building block it is easier to control the concurrency throttling of Worker nodes, enabling this feature for the A2A requires additional coordination among L-Workers and currently it is not implemented.

The A2A building block has been thought mainly to optimize *farm* configurations where the Emitter node can be easily replicated, and in general, where the Emitter needs to be parallelized to avoid potential bottlenecks. The A2A avoids introducing any centralized entity, thus enabling higher scalability than the farm even though at higher costs in terms of complexity management.

7.3.4 Evaluation

In this section we evaluate the raw performance of the *farm* and *all-to-all* building blocks considering two simple micro-benchmarks. The experiments reported in this section, were conducted on the *Xeon* and *KNL* platforms (see Table 6.1).

***farm* building block** The first test is a simple farm with the Collector. The Emitter generates $100,000$ data elements toward the pool of Workers as fast as possible. The channel's capacity is fixed so that the Emitter cannot produce more that $10,000$ elements to the Workers. The Collector just receives results produced by the Workers. The service time of the generic Worker of the pool is fixed to a constant value varying from, very fine-grained granularity $1\mu s$, fine-grained granularity $10\mu s$, and medium-grained granularity $100\mu s$.

The average results, over 10 executions, obtained running this simple benchmark is reported in Figure 7-10 (the standard deviation is less than 1%). On the *Xeon* platform even for very fine grained computation granularity the *farm* building block can obtain almost a linear scalability up to 16 Workers. This is not the case on the *KNL* platform where $1\mu s$ computation granularity (about 1500 clock cycles) is

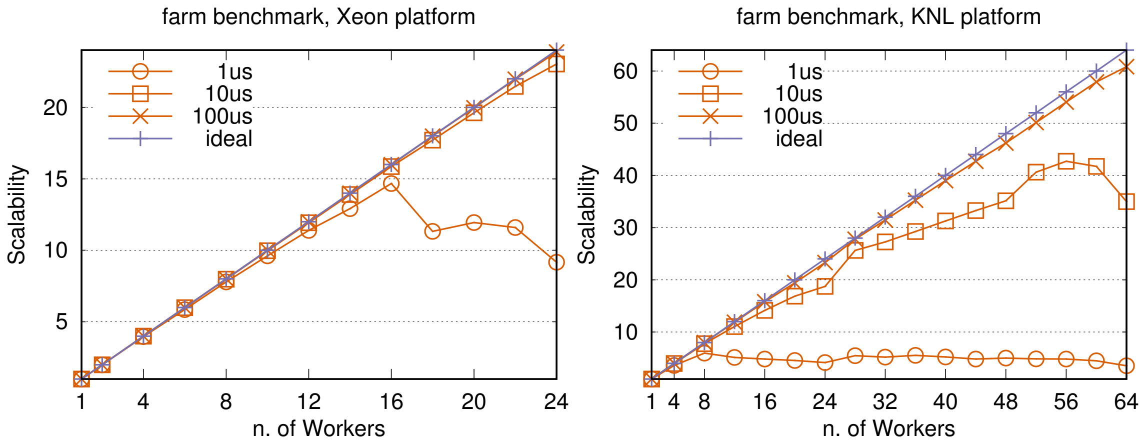


Figure 7-10: Farm scalability varying the computation granularity of the Worker function on the *Xeon* (left-hand side) and on the *KNL* (right-hand side) platforms.

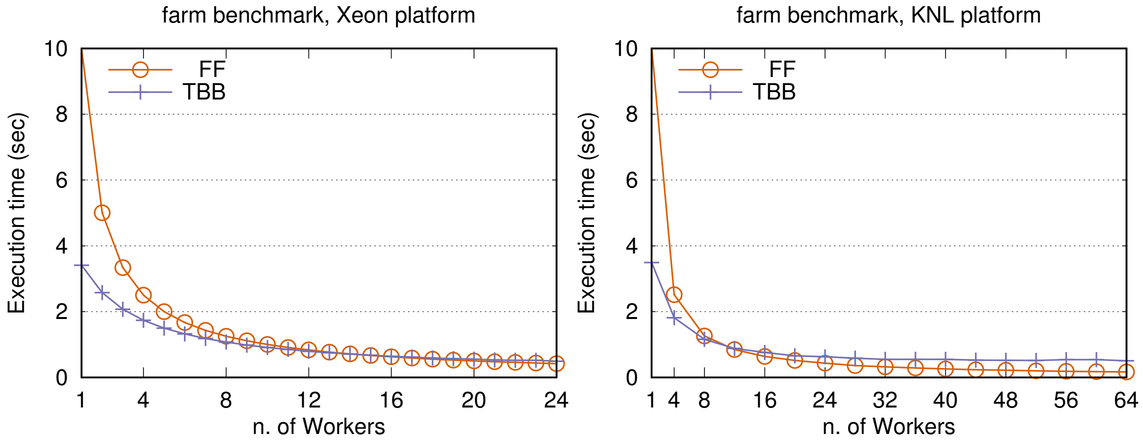


Figure 7-11: Farm benchmark implemented with FastFlow and Intel TBB on the *Xeon* (left-hand size) and on the *KNL* (right-hand side) platforms.

too small to obtain good speedup figures. Increasing the granularity of one order of magnitude allows us to obtain a scalability of about 43 with 56 Workers (and 58 threads).

The same test has been implemented using Intel TBB by using the *parallel pipeline* construct provided by the library[1]. It can be used to model a *Task-Farm* pattern. For the benchmark considered it implements a three-stage pipeline. Specifically, the first and last stages (corresponding to the Emitter and the Collector) are *sequential filter* whereas the middle stage is a *parallel filter* whose maximum parallelism degree

[1] The TBB *parallel pipeline* allows us to execute multiple stages in parallel

is based according to the number of threads instantiated for the TBB run-time (in our test it is equal to the number of Workers plus two). The maximum in-flight token for the Intel TBB run-time has been set to $20,000$. The TBB library used is the one shipped with Intel Parallel Studio XE 2017. The results obtained are shown in Figure 7-11.

As can be seen, for a low number of Workers the TBB version provides the best execution time thanks to its work-stealing based scheduler that is able to better utilize the available threads in the run-time. Then, by increasing the number of Workers, the FastFlow RTS is able to obtain slightly better results because of its direct mapping between nodes and RTS threads. On the *Xeon* platform the lowest execution time for the FastFlow and TBB versions is 418ms and 489ms, respectively. On the *KNL* platform it is 165ms and 505ms, respectively. This simple benchmark clearly shows that the FastFlow *farm* building block is effective for implementing the *Task-Farm* pattern and its implementation provides good scalability figures even for fine-grained computations.

***all-to-all* building block** To test the *all-to-all* building block, we used the parallel schema sketched on the right-hand side of Figure 7-12 (1-1 configuration). It is a four-stage pipeline where the first and last stages are a *multi-output* node and a *multi-input* node, respectively. They act as stream source and stream sink and do not execute any business logic. The first node injects $100,000$ data items into the pipeline, and the benchmark terminates when the last stage collects all data items.

The two middle stages (second and third stages) execute a synthetic computation. They have been configured such that the second stage has a service time that is three times larger than the third stage (e.g., $30\mu s$ and $10\mu s$). Our objective is to parallelize the second and third stage with an A2A building block and to measure the overall performance of the pipeline. To balance the service times, the number of Workers in the L-Worker set is three times larger than the number of Workers in the R-Worker set. In this way, the two set of Workers have about the same service time in the ideal case of zero overhead for the parallel implementation.

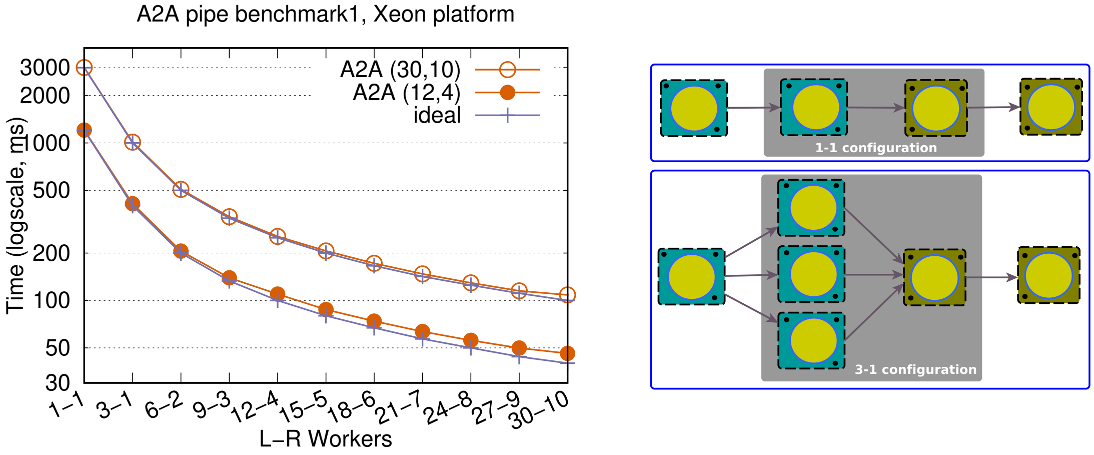


Figure 7-12: A2A pipeline benchmark. *Left)*: total execution time varying the number of L- and R-Workers proportionally to their service time T_s ($T_s^{R-Workers} = 30\mu$s $T_s^{L-Workers} = 10\mu$s). *Right)*: parallel schemas of two configurations tested: 1-1 and 3-1.

In Figure 7-12 is shown the execution time varying the number of Workers in the L- and R-Worker set and considering two different cases for the service time: $30\mu s$, $10\mu s$ and $12\mu s$, $4\mu s$ corresponding to the service time of the generic L-Worker and R-Worker, respectively. As can be seen, the execution time has an almost perfect decrease when the number of Workers of the A2A component is increased proportionally. This simple benchmark demonstrates that the FastFlow implementation of the A2A building block introduces a negligible overhead in the pipeline execution of the L- and R-Workers.

To study the scalability of the *all-to-all* building block we used the same benchmark described before (a four-stage pipeline). However, differently from the previous case, we considered the following configurations: 1) Workers of the first and second set of the A2A have the same service time (fixed for the given platform); 2) only the number of L-Workers are varied while the number of R-Workers is kept fixed for the entire test. For the execution of the A2A benchmark on the *Xeon* platform we used a service time of $1\mu s$ and 12 R-Workers. For the *KNL* platform, the service time is 10 times higher while the number of R-Workers is fixed to 32. The results obtained are shown in Figure 7-13. On the *Xeon* platform, the A2A exhibit linear scalability up to 10 R-Workers then the curve starts flattening. It is worth pointing out that, with a

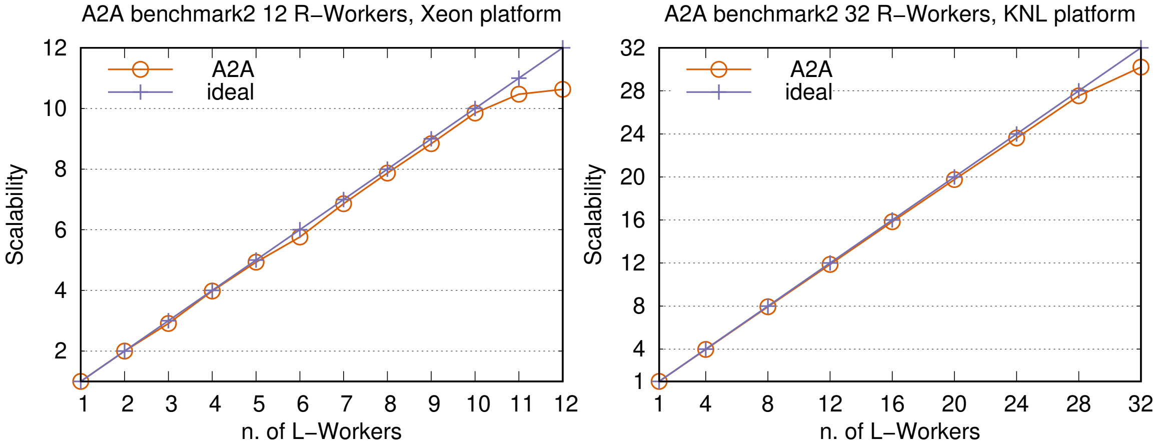


Figure 7-13: Scalability results for the A2A benchmark. *Left)*: *Xeon* platform, $T_s =$ 1us and the number of R-Workers is 12. *Right)*: *KNL* platform, $T_s =$ 10us and the number of R-Workers is 32.

service time of $1\mu s$ for the single Worker, the service time of the R-Workers is about 100ns, therefore close to the time needed for a single point-to-point communication for the considered platform (see Chapter 6). On the KNL platform, the scalability trend is close to the ideal one demonstrating that the building block implementation does not introduce significant overheads.

As for the *farm* building block, also for the A2A component, the simple tests proposed in this section demonstrate good performance figures even for fine-grained computations.

7.4 Preserving stream ordering

In streaming computations, preserving stream ordering is an important feature of many applications (e.g., video processing). Maintaining data ordering does not necessarily mean to preserve total ordering among elements, sometimes this constraint can be relaxed to have only a partial ordering, such as in *key-based* computations where ordering is usually kept only among elements having the same key [31].

Data ordering is a critical issue that could have a serious impact on the overall application performance. Usually, preserving ordering requires buffering of data elements arrived out-of-order and thus requires to establish how much memory to reserve for the buffer.

While the pipeline of sequential building blocks naturally preserves data ordering, this is not the case if some stages of the pipeline are implemented with a *farm* and/or with an *all-to-all* building block. In the following, we briefly discuss how the FastFlow library deals with data element ordering when these two building blocks are used.

Ordered farm In the FastFlow library, farm's Workers are executed within separate threads of execution. Therefore there is no guarantee that the output sequence produced by the Workers respects the input ordering of data elements as seen by the *Emitter*. Due to the relative speed of each Worker and due to the different execution time associated to different data elements, the results produced by the Workers might arrive at the Collector in an unpredictable order. By default, the *farm* building block does not preserve input ordering because in many applications ordering is not needed and preserving data ordering might introduce extra overhead not worth paying if not required. If preserving input ordering is essential for the application at hand, as in parallel video processing applications where the order of frames must be maintained in the output video, then the *farm* building block may enforce input-output ordering if the user explicitly calls the method `set_ordered`. In such a case, a default Collector is automatically added by the RTS if the user will not provide one.

The most straightforward and most efficient way to enforce data ordering in the farm building block is to schedule data elements in a given fixed order and to gather Workers' results in the Collector following the same order. This simple protocol does not require any extra buffering in the Collector and introduces only a minimal overhead. However, if the execution time associated with different input data elements has a high time variance, this simple strategy does not provide the best performance due to potential workload imbalance introduced by the static scheduling and gathering policies.

As pointed out in Dalvan et al. [178], to solve the issue of workload imbalance in an ordering FastFlow farm, a simple solution would be to tag data elements when they enter the Emitter node and by buffering out-of-ordered elements in the Collector on the basis of their tag which has been forwarded by the Workers. The workload can then be balanced by using a dynamic scheduling policy such as the *on-demand* policy (see Section 7.3.2). We have implemented this policy in FastFlow-3. If both the methods `set_ordered` and `set_scheduling_ondemand` are called on a farm object, the FastFlow RTS *automatically and transparently to the user* adds a unique tag to each data element entering the farm. This also implies that the Workers nodes are wrapped into system nodes that remove the tag before calling the service functions, and re-add the same tag right before the result is sent to the Collector.

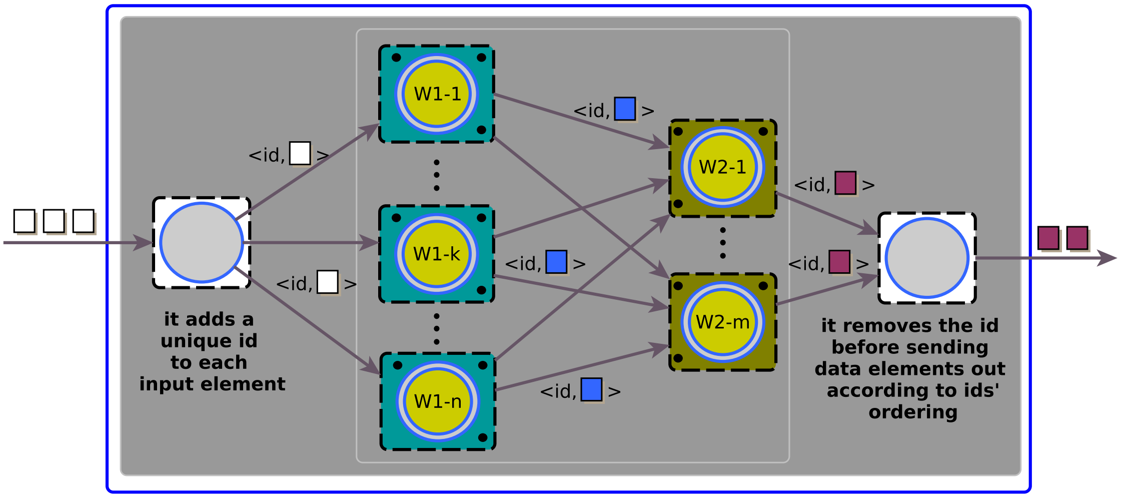
However, enforcing stream ordering for the *farm* building block introduces some limitations:

1. Workers can be implemented using only standard nodes;
2. Workers cannot use in their service functions the method `ff_send_out` and the `svc` method cannot return the special value `GO_ON`.

While the first point is mainly a limitation of the current implementation of the library and therefore can be relaxed in the next versions, the second point is critical because the stream length cannot be increased/decreased by the Worker nodes otherwise the ordering policies described before does not work. For these unusual cases, the user has to provide his/her application-specific solution and can simply use the standard farm building block and the mechanisms offered by the FastFlow nodes.

Ordered All-to-All Maintaining the input data element ordering for the *all-to-all* building block is a feature that is not natively provided by the component. That is due to the lack of a single entry point for all data elements in the `all-to-all` building block.

A simple solution would be to encapsulate the *all-to-all* building block as a unique Worker of a farm where the Emitter and Collector entities maintain data ordering as described in the previous sections. This operation is not as straightforward as it seems at first glance. The default round-robin scheduling policy of the farm does not guarantee data ordering because of the multiple connections between L-Workers and R-Workers. Therefore, without any extra information, the Collector node cannot



W1-1
W1-k
W1-n
W2-1
W2-m
<id, □>
<id, □>
<id, ■>
<id, ■>
<id, ■>
<id, ■>
<id, ■>
it adds a unique id to each input element
it removes the id before sending data elements out according to ids' ordering

Figure 7-14: Logical implementation schema of an *ordered all-to-all* building block.

know which is the correct order to follow in the gathering of data. Therefore, regardless of the data scheduling policy of the farm, the only solution is to use the policy which tags input data elements with a unique monotonic identifier.

To alleviate the user from the burden of dealing with Worker wrapping and other boilerplate code, the FastFlow framework provides a transformation function that, given in input an *all-to-all* building block having standard nodes as L-Workers and R-Workers, returns a pipeline containing an order-preserving farm whose Worker is the *all-to-all* building block provided in input. As described before, the Emitter of the order-preserving farm tags each input element with a unique identifier while the Collector buffers out-of-order data elements until the correct values can be sent out to the next stage. The logical schema of the "*ordered* `all-to-all`" is sketched in Figure 7-14.

7.4.1 Evaluation

To evaluate the overhead introduced by the automatic ordering feature of the FastFlow farm building block, we considered two video applications: i) Lane Detection, and ii) Face Recognition [105]. Both applications have been implemented by using a farm building block and defining the Emitter and Collector nodes to read the input video from the disk and to write the output video into the local disk, respectively.

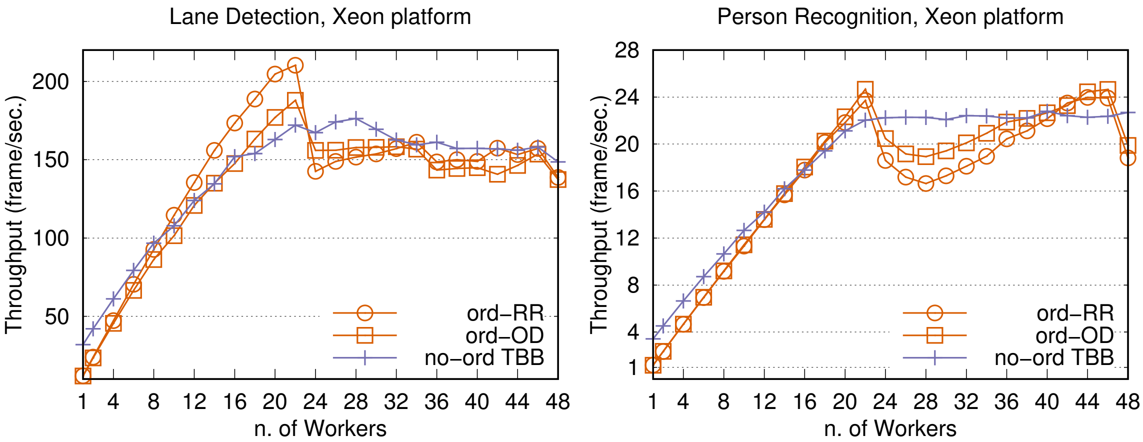


Figure 7-15: Comparison of the throughput for the Lane Detection (left-hand side) and Person Recognition (right-hand side) applications obtained by different some FastFlow ordering policies vs a not-order-preserving Intel TBB version.

The logical parallel structure of the two applications is that of a three-stage pipeline (read-compute-write). Since video frames ordering has to be preserved, the farm building block is configured to be an ordered farm (i.e. the *set_ordered* method was invoked on the farm object). The experiments were conducted on the *Xeon* platform (see Table 6.1).

Lane Detection (LD) is a video application used on autonomous vehicles to detect lane lines on the road. This application reads the video from an onboard camera recording the whole lane and detects road lines in real-time. *Face Recognition* (FR) is an application used in video controlled systems to prevent unauthorized accesses in a room. Even in this case, FR reads data frames from a camera and performs the face recognition in real-time using an image database.

For LD and FR applications we considered both a FastFlow implementation using an ordered farm building block and a TBB version as baseline that does not preserve frames ordering (and thus without additional overhead due to the ordering). For the FastFlow version, we tested both the default round-robin scheduling policy (ord-RR) and the on-demand scheduling policy (ord-OD) with an asynchronous degree of 8 frames. The maximum throughput obtained varying the number of Workers are shown in Figure 7-15 (the number of threads used by the TBB run-time is equal to

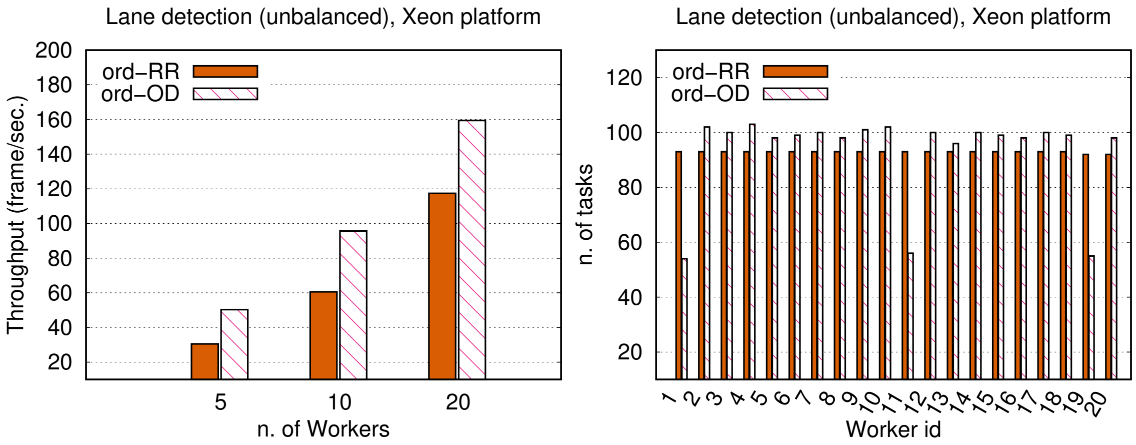


Figure 7-16: *Left)*: Maximum throughput of the modified Lane Detection application to produce unbalanced workload among *farm* Workers. *Right)*: Number of video frames computed by each Worker.

the number of Workers plus two while the number of in-flight tasks was set to a value larger than the number of frames).

For the LD application, the maximum throughput is obtained by the FastFlow version with the round-robin scheduling policy and 22 Workers (for a total of 24 system threads). The configuration ord-OD is not able to obtain the same performance results due to the higher overhead introduced by the automatic data element tagging and to a relatively low variation in the per-frame service time. The average per-frame service time is about 87ms with a standard deviation of about 20ms. The TBB version obtains the lowest maximum throughput even if it exhibits a smoother behavior when more threads than cores are used (i.e. starting from 22 Workers).

Concerning the FR application, it has a higher service time and a higher standard deviation, i.e. about 880ms and about 195ms, respectively. In this case, the ord-OD version obtains the highest throughput (24.7 vs 23.7 obtained by using the ord-RR version).

To test the performance differences of the ord-RR and ord-OD, we artificially increase the service time of the Lane Detection application. Specifically, 3 randomly selected Workers execute extra works for each input frame. The new average service time of the LD application is about 99ms (vs. 87 of the original version) with a standard deviation of about 40ms. We considered the case with 5, 10, and 20 Workers. In the left-hand side of Figure 7-16 are reported the maximum throughput obtained. In the right-hand side of the same figure we reported the number of frames computed by each Worker for the two scheduling policies. As expected, the ord-OD provides a better distribution of frames to Worker compared to the ord-RR which assigns almost the same number of frames to each Worker.

7.5 Summary

In this chapter we described the FastFlow implementation of the sequential and parallel building blocks introduced in Chapter 5. By using simple micro-benchmarks modeling streaming computations, we assessed the overhead introduced by the FastFlow implementation of the building blocks. Specifically, we tested the sequential *nodes combiner*, the *farm* and the *all-to-all* building blocks. The *pipeline* is the base building block used for all tests. The results obtained on two different multi-core platforms (a standard Intel Xeon-based multi-cores, and the Intel KNL many-cores) demonstrate that the implementation proposed introduces low run-time overheads and good scalability. We also discussed the important problem of maintaining stream ordering. The solution proposed for the *farm* building block has been assessed by using two video applications. The results obtained demonstrate that the solutions proposed introduce only a minimal overhead and provide good performance figures also in the case of unbalanced workloads.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Concurrency Graph Transformations

8.1 Introduction

In this chapter, we present the new FastFlow software layer providing a set of mechanisms and functions that support the static restructuring of the application concurrency graph. The primary objectives of graph transformations are both to refactor and to optimize the application graph by reducing the number of FastFlow nodes and by optimizing particular combinations of building blocks. We are interested in those transformations that maintain the functional semantics of the application while increasing the throughput and/or decrease the latency of data elements flowing into the graph.

From the theoretical standpoint, graph's transformation and rewriting is a well-known and widely studied problem (see for example [287]). In this chapter we do not study the general problem, we focus on a restricted set of graphs that can be derived from the composition of FastFlow building blocks with the aim of studying their non-functional properties.

Given the direct relationship between the nodes of the FastFlow concurrency graph and the actual mapping of nodes to system-level threads (cf. Section 4.3), the refactoring and optimizations rules are the primary mechanisms for promoting and enhancing *performance portability* of FastFlow applications on current and forthcoming highly-heterogeneous multi-cores.

Thanks to a clear functional and parallel semantics of the building blocks, several different transformations can be developed that allow the programmer to transform a FastFlow program into a functionally equivalent one achieving different performance levels (both regarding latency and bandwidth) and using a different number of system resources. These transformations can be driven by some analytical performance models associated to the concrete implementation of the building blocks on the given platforms with the aim to select only those transformations leading to efficient implementations according to some target function [63].

However, in the following, we do not propose an analytical model nor possible transformation strategies. Instead, we describe how we enriched the FastFlow library with the essential mechanisms needed to implement the most interesting transformations and how such transformations are implemented. Policy-oriented software layers, higher-level frameworks or DSLs can be built on top of the FastFlow graph transformation software component thus benefiting from the mechanisms provided by its API to devise smart transformation policies possibly guided by dynamically collected performance metrics [131].

A simple interface function (called `optimize_static`) is provided to the FastFlow user to automatically apply straightforward transformations to the pipeline compositions describing the application (or even parts of it). In the context of the FastFlow concurrency graph transformations, the word *automatic* means "user-suggested and automatically applied by the RTS". The programmer may ask the FastFlow RTS to automatically modify the concurrency graph, for example, to remove all default Collector nodes in the *farm* building blocks. In order to preserve the functional semantics of the application, the transformations will be implemented only if a predefined set of conditions hold, otherwise, the concurrency graph will not be modified. The programmer may also choose to use ready-to-use optimizations (implemented by the `OptLevel` object) that enable a subset of specific transformations automatically. They can also explicitly select which transformations have to be automatically implemented for the given pipeline composition by simply setting specific flags in the `OptLevel` object. However, the expert programmer, instead of relying on simple automatic transformations, may use the available API to "manually" apply specific and more complex building block transformations to the *pipeline* or *farm* passed as arguments, which might implement only a part of the entire application. These transformations can, for example, by guided by some sophisticated static analysis on data collected from previous executions [62]. Currently, the most complex and potentially critical graph transformations from the functional semantics viewpoint, are in charge of the application/RTS programmer. In other words, the *concurrency graph transformer* software components provides a broad set of mechanisms and helper functions that can be used by the expert programmer to decompose and rebuild the concurrency graph of FastFlow applications for the sake of introducing optimizations.

The automatic graph transformations currently provided to the user through the optimization interface are:

1. Removing default Collector nodes from the *farm* building block and from its nesting instances into parallel building blocks.
2. Combining a sequential node with the subsequent *farm*'s Emitter.
3. Merging sequences of *farm*s with the same number of Workers (*farm fusion*).
4. Combining two *farm*s with non-default Emitter and/or Collector nodes (*farm combine*).
5. Combining two *farm*s by using the *all-to-all* building block (*all-to-all introduction*) if the Collector of the first farm and the Emitter of the second farm are default service nodes.

The rest of this chapter is organized in two parts. The first part introduces the graph transformations currently implemented and also describes which ones can be implemented automatically and under what conditions. The second part focuses on the evaluation of some of the transformations proposed by considering applications and use-cases implemented in other libraries or frameworks (e.g., PiCo and `WindFlow`) that are using the FastFlow library as RTS.

8.2 Process Network Transformations

The directed FastFlow application graph is composed of different kinds of sequential nodes connected by point-to-point channels. Nodes' functions, either executing business logic or RTS code, are executed in parallel on the available cores according to the data-flow execution model. The graph represents the process network topology describing the parallel application, and it can be analyzed and statically transformed by combining or removing nodes. Apart from simple compositions of two *sequential* nodes that can be obtained by using the *sequential node combiner* building block (described in Section 7.2.3), other compositions might not be simple to implement for the programmer. For this reason, the FastFlow framework provides a *concurrency graph transformer* software component which implements a set of functions that can be used to combine parallel building blocks in several different ways with the aim to apply interesting and useful transformations to the process network topology. In the following, we discuss some of these functions covering the most notable cases.

When the *farm* building block is connected in pipeline with other building blocks (either sequential or parallel), the Emitter and Collector entities might be used just to set proper input/output cardinality and to provide the connections with previous and subsequent building blocks. In these cases, they are "service nodes" and do not execute any business logic. However, the FastFlow RTS reserves to them a dedicated thread. The Emitter node cannot be removed because the base *farm* building block requires it as a master node. Instead, the default Collector node can be removed forcing the Workers of the *farm* to send the output results directly to the next building block. However, even if the default Emitter cannot be removed, it can be combined with a previous sequential node (if present) and, from the viewpoint of the resource usage, this operation is semantically similar to the Emitter removal.

We distinguish four transformations aimed at reducing the number of service nodes or at removing potential bottlenecks:

1. Given a single *farm* building block, the transformation removes the Collector node (if present) and combines the previous sequential node (if present) with

the default *farm* Emitter. These transformations are described in Section 8.2.1.

2. Given a sequence of *farm* building blocks, the transformation fuses all *farm*s in a single *farm* building block. This transformation is described in Section 8.2.2 (*farm fusion*).

3. Given two *farm*s in pipeline that cannot be merged (for example because they have two different scheduling policies), the transformation combines them removing the Collector of the first farm. This transformation is described in Section 8.2.2 (*combining two farms*).

4. Given two *farm*s in pipeline, the transformation substitutes the two *farm*s with a single *farm* whose Worker is an *all-to-all* building block. The objective is twofold: i) to remove the intermediate Collector and Emitter nodes; ii) to replicate one or both of them exploiting the *all-to-all* parallel structure. This transformation is described in Section 8.2.3.

For the sake of simplicity, unless stated otherwise, in the rest of this chapter we will examine cases where *farm*'s Workers and *all-to-all*'s Workers are sequential building blocks.

8.2.1 Reducing service nodes

Let us consider the case sketched in Figure 8-1 where the application (or part of it) is a three-stage pipeline composed by two *standard sequential nodes* (the first and last one) and a *farm* building block with default Emitter and Collector nodes.

This network can be transformed in one of the two options depicted at the bottom part of the figure (i.e. opt1 and opt2). Both configurations have the same parallel semantics of the original topology even if they have a lower concurrency degree. From the standpoint of the FastFlow RTS, the two configurations allow the programmer to spare two threads. This transformation can be applied by calling the optimization function `optimize_static` and setting the flags `remove_collector` and `merge_with_emitter` in the *OptLevel* object passed as argument to the function. As

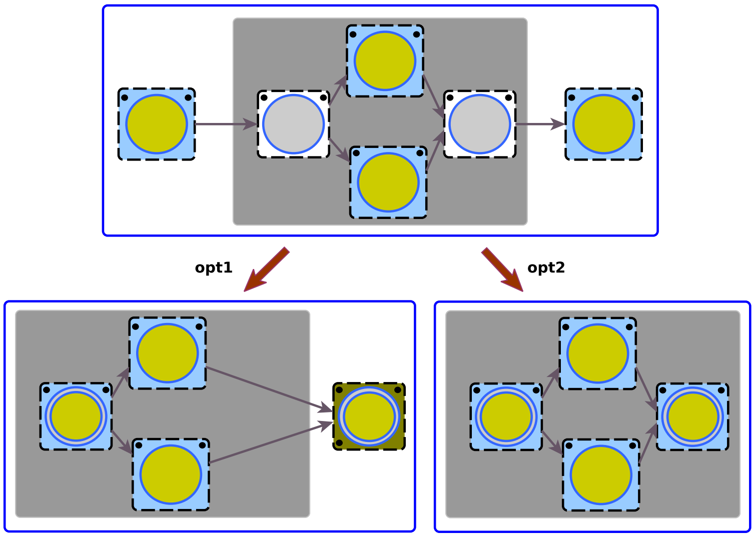


Figure 8-1: Transformation that removes the default Collectors and merges previous stage with *farm*'s Emitter.

```
 1 ... // creating Stage1, myFarm and Stage2
 2 ff_Pipe<> pipe(Stage1, myFarm, Stage2);
 3
 4 OptLevel opt;
 5 opt.remove_collector  =true;
 6 opt.merge_with_emitter=true;
 7 optimize_static(pipe, opt);
 8
 9 .... // adding the transformed pipe to another pipe
10 ff_Pipe<> pipe2(Stage0, pipe, Stage3);
11 ...
12 pipe2.run();
```

Code 22: How to use the `optimize_static` function.

an example, let us consider the snippet of Code 22 where the farm Collector is removed (line 5) and the `Stage1` is merged with the farm's Emitter of the `myFarm` instance (line 6). The transformed pipeline (`pipe`), obtained as result of the `optimize_static` function (line 7) is then added as a middle stage of the second pipeline `pipe2` (line 10).

As we shall see, the `optimize_static` function can be used to apply multiple transformations using the same approach.

Another interesting case from the viewpoint of reducing the size of the FastFlow concurrency graph is the nesting of *farm* of *farms*, i.e. a *farm* building block whose

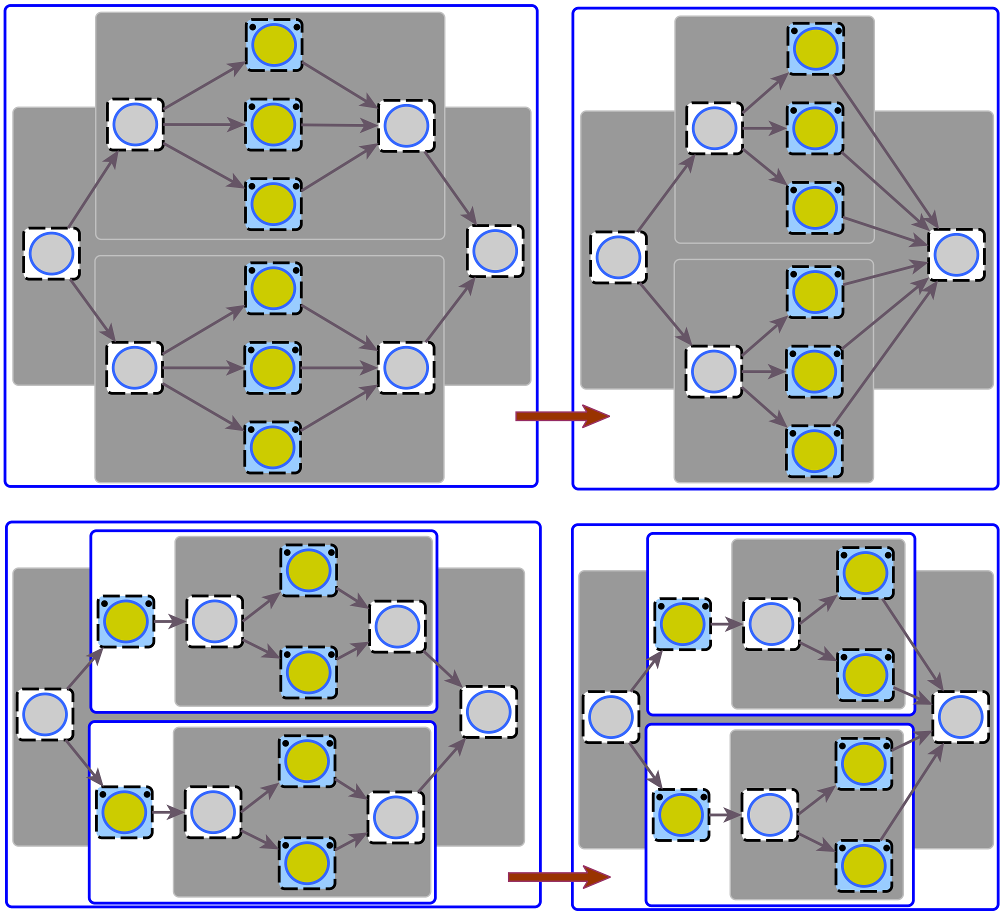


Figure 8-2: *Farm* of *farms* transformations. The default *farm*'s Collector of the internal *farms* are removed.

Workers are *farms* as well. Even though *farm* nesting is not so common, it can be the result of automatic code refactoring and/or the application code rewriting rules [163].

If the Collector nodes of the internal *farms* are default Collectors, then it is possible to remove them. All results produced by Workers of internal *farms* will be collected by the outermost *farm* building block (see Figure 8-2). This transformation is automatically implemented by the FastFlow RTS by calling the function `optimize_static` (and setting the flag `remove_collector`) either on the top-most *pipeline* building block or directly to the *farm* building block that the user want to refactor. The implementation recursively looks for *farm* contained in a *farm* building block. A particular case handled by the *optimize_static* function is when the internal *farm* (with a default Collector) is the last stage of a *pipeline* building block used as Worker of a *farm*.

Other interesting opportunities in which it is possible to reduce the service nodes arise when there are two *farm*s connected in pipeline. We will discuss these cases in the next sections (Section 8.2.2 and Section 8.2.3).

8.2.2 Farms in pipeline

In this section we want to analyze the problem of *combining* and *fusing* two or more *farm* building blocks. We distinguish between *combining two farms* and *fusing two farms*. In the first case only a few parts of the two building blocks are merged. Therefore, after the transformation, there are again two instances of the *farm*. In the second case the two instances are merged into a single *farm* instance. Fusion is a technique in which a sequence of operators, executed by a concurrent entity and forming a pipeline graph, are merged and executed by a dedicated concurrent entity.

The motivations for implementing the fusion and the combine operations for two (or more) *farm* building blocks, may be several:

1. To reduce the number of active entities that are used in the RTS implementation of two *farm*s in sequence.

2. To reduce the number of hops (i.e. the number of channels and nodes) a data element has to traverse from when it enters the first *farm* to when it exits from the second farm.

3. To avoid potential bottlenecks introduced by the communication channel between the Collector of the first *farm* and the Emitter of the second *farm*.

4. To eliminate potential bottlenecks introduced by either the sequential Collector of the first *farm* or by the sequential Emitter of the second *farm*.

In the following we examine the first two cases. The other two will be discussed in Section 8.2.3.

Algorithmic skeleton transformations have been extensively studied in the past [239, 14]. Some skeleton-based frameworks already implement the fusion operation mainly applied to data parallel skeletons (the so called *map fusion*) [148, 61, 147]. Other non-skeleton approaches such as StreamIt [333], uses the filter fusion operation to coarsen the granularity of streaming filters based on cost estimate [172]. From the theoretical standpoint, it has been demonstrated that any stream parallel composition of *Pipeline* and *Task-Farm* patterns can always be transformed into an equivalent "normal form" skeleton (i.e. a single *farm* building block whose "fat" Workers execute multiple functions), which uses the same resources (in a different way) and delivers at least the same service time [14].

On the basis of these theoretical and practical results, we are interested in providing the FastFlow user with mechanisms that allow fusing pipeline sequence of *farm*s regardless of what they are used for, i.e. for implementing a *Task-Farm* or a *Map* patterns.

Farm fusion

The *farm fusion* transformation can be performed without affecting the result computed by the FastFlow program if the Emitter and Collector entities are default RTS nodes. It simply changes the amount and kind of parallelism in the concurrency graph describing the application, hence it may affect the performance potentially achieved by the application. Examples of computations that use sequences of multiple *Task-Farm* patterns are *Ferret* [228] and *Dedup* both part of the PARSEC benchmark suite targeting multi-core platforms [47]. We will study these two applications in the evaluation section (Section 8.3).

The function `combine_farm_nf`, part of the FastFlow transformation API, gets in input two *farm*s with sequential Workers and returns a new *farm* building block whose Workers are obtained by using the *sequential node combiner* building block that merges Workers of the first *farm* with the corresponding Workers of the second *farm*. The function can be called multiple times to combine more than two *farm*s. The only requirements for calling the `combine_farm_nf` function is that the two farms have the same number of Workers and that both *farm*s are either "standard" or "ordered" *farm* building blocks. If the two *farm*s are *ordered farm*s then to be fused they must have

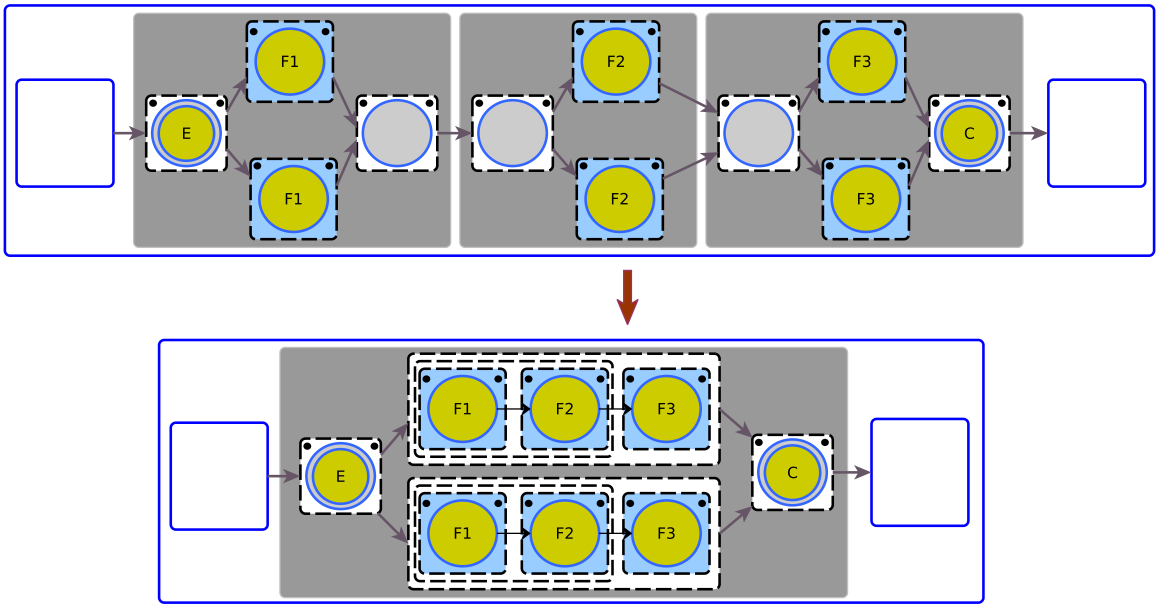


Figure 8-3: *Farm fusion* transformation

the same ordering policy. No other checks are performed by the function. This means that, if the first *farm* has a non-default Collector and/or the second *farm* has a non-default Emitter, the transformation is discarded. Therefore, this function has to be properly used by the RTS programmer, typically in conjunction with other low-level transformation functions to obtain the correct final *farm fusion* transformation.

However, under particular conditions, the *farm fusion* transformation is implemented automatically by the optimization function `optimize_static`. The programmer does not need to call the function `combind_farm_nf`. The function `optimize_static` accepts the flag `merge_farms` set in the *OptLevel* object passed as parameter to the function (see also Code 22). Its implementation looks for the longest sequence of *farms* having all the requirements necessary to enable the application of the *farm fusion* operation, and then directly modifies the FastFlow concurrency graph substituting the sequence found with the new transformed version. The requirements for the "automatic" *farm fusion* are the following:

1. The sequence of *farms* must have the same number of Workers.
2. The *farms* must use default Emitter and Collector entities (the Collector may not be present). The first farm of the sequence may use a custom Emitter node

and the last farm of the sequence may use a custom Collector node.

3. The channel configuration must be the same for all *farm*s (i.e. all farms must use either bounded or unbounded channels).
4. The data scheduling strategies must be the same (either round-robin or on-demand).
5. If the first *farm* of the sequence is an order preserving *farm*, then all other *farm*s must be order preserving *farm* using the same policy.

.

Figure 8-3 shows the result of the *farm fusion* operation applied to a pipeline containing a sequence of three *farm*s.

Combining two farms

When it is not possible to fuse two *farm*s because they have a different number of Worker nodes or because the Collector of the first farm and/or the Emitter of the second farm are non-default ones, it might still be possible to combine them. We call this operation *farm combine* transformation.

Let us consider the cases sketched in Figure 8-4. The figure shows three cases where the *farm fusion* transformation cannot be applied. However, it is possible to combine the Collector of the first *farm* with the Emitter of the second *farm* building block regardless of whether they are user-defined or not. In this way, the sequence of the two *farm*s spares a FastFlow node.

All transformations sketched in Figure 8-4 are implemented by the function `combine_farms` (together with other transformations that will be discussed with more details in Section 8.2.3). The function gets in input two *farm*s to combine and returns a *pipeline* containing the transformed building blocks.

An interesting case of the *farm combine* transformation is given when in the sequence of two *farm*s there is an ordered *farm* building block (i.e. it guarantees that the output sequence produced respects the input ordering of data elements as seen by the Emitter – Section 7.4). This case introduces some constraints given by the

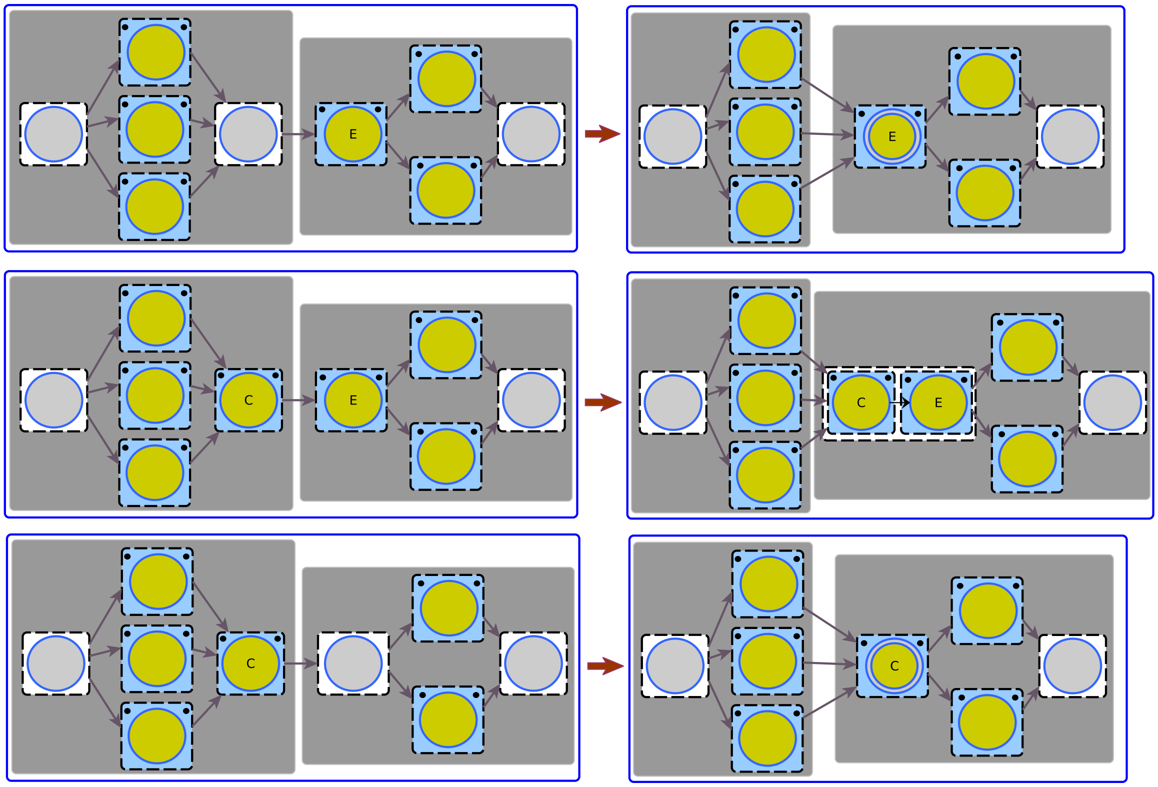


Figure 8-4: *Farm combine* transformations.

need to preserve the ordering of stream elements.

The transformations allowed in this case are the following ones:

1. The first farm is a standard *farm* building block while the second one is an ordered farm. This is the simplest case, if the Collector of the first farm is present, then it can be removed, otherwise it can be merged by means of the *sequential node combiner* building block with the Emitter of the ordered farm.

2. The first farm is an ordered *farm* building block while the second one is a standard *farm*. In this case, the Collector of the first farm cannot be removed even if it is a default Collector because its presence is required to preserve the ordering of stream data elements. The transformation builds a new *pipeline* composed by two *farms* where the first farm has a new Emitter which adds unique ids to each input element produced toward the Workers (it also wraps the user code executed by the Emitter of the first *farm*, if it were present). The user code executed by the Workers is wrapped in a RTS node which removes the

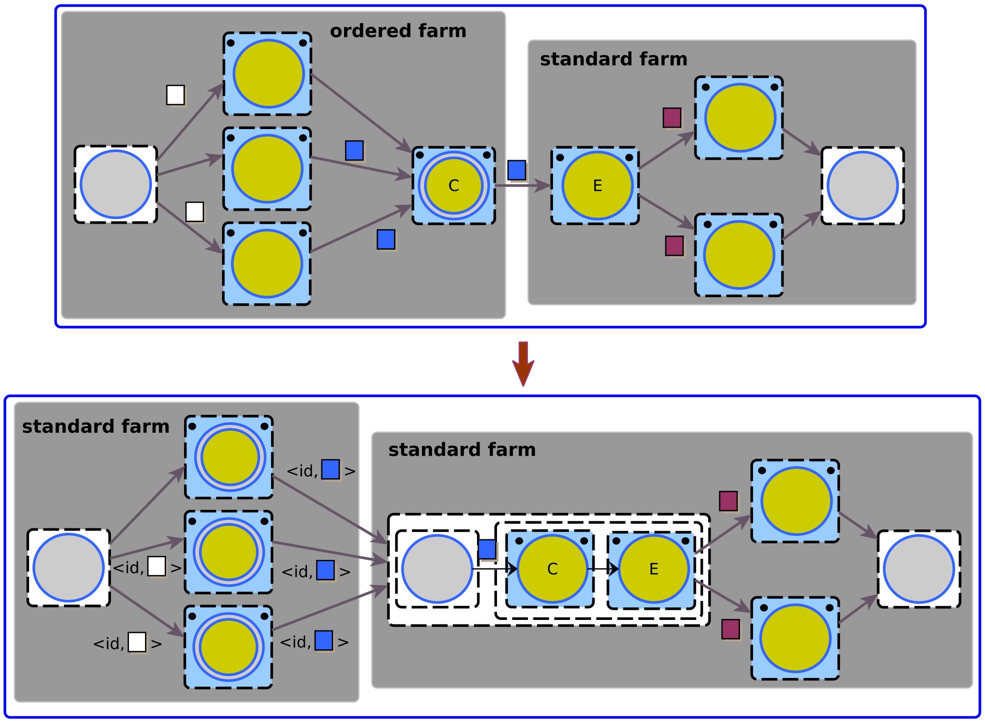


Figure 8-5: Combining *ordered farm* and standard *farm* in pipeline.

id before calling the `svc` method of the user's node and then adds again the same id before the result is forwarded to the Emitter of the second *farm*. The second *farm* has a new Emitter which is a composition of two sequential nodes: the first one is a RTS node which on the basis of data elements' ids produces them in output in order (without the ids); the second one is a *sequential combiner node* between the Collector of the first farm (if present) and the Emitter of the second *farm*. The resulting topology produced by this transformation is shown in Figure 8-5.

The first transformation is handled as a particular case of the `combine_farms` function. The second case is handled by the `combine_ofarm_farm` function which gets in input the two *farms* and returns the transformed *pipeline* building block.

Currently, the second transformation is not provided by the `optimize_static` function described in Code 22.

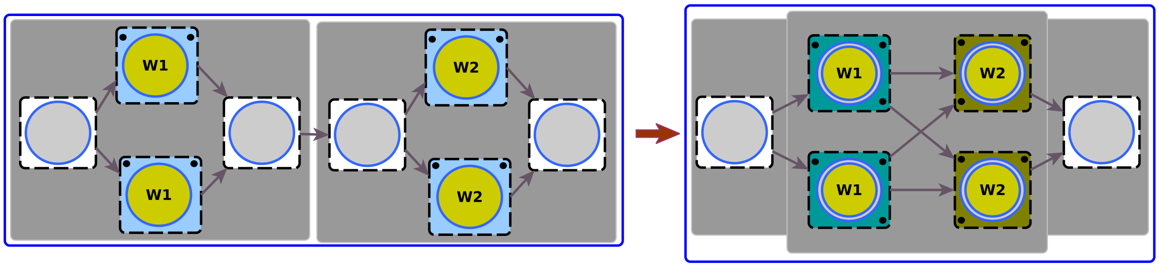


Figure 8-6: Combining two *farms* by using the *all-to-all* building block.

8.2.3 All-to-all introduction

As we have seen in Section 8.2.2, given two *farms* in pipeline, if both the Collector of the first *farm* and the Emitter of the second *farm* are default nodes, then it is possible to apply the *farm fusion* transformation. Instead, if one of the two or both are user-defined nodes, then we can apply the *farm combine* transformation.

Here we consider an alternative transformation. Instead of fusing the Worker nodes, we want to combine them by using the *all-to-all* building block. The primary objective of this transformation is not to reduce the number of RTS threads. Instead, it is to remove the potential bottleneck introduced by the Collector and Emitter service nodes without sacrificing the amount of parallelism of the application. The new topology is sketched in Figure 8-6. The Workers of the first and second *farm* are automatically transformed into *multi-output* and *multi-input* nodes, respectively. The Workers of the first *farm* are added to the L-Worker set of the *all-to-all* while the Workers of the second *farm* are added to the R-Worker set of the same *all-to-all* building block.

If the Collector of the first *farm* and/or the Emitter of the second *farm* are user-defined, the two *farms* can be combined without any application change if both of them could be replicated. This means that they do not have any internal state or that the internal state is used read-only. The resulting topology is the one that is shown in Figure 8-7 where `R=E` and `G=C`. In the general case, the nodes `R` and `G` can be any sequential composition of nodes such that they are eventually a *multi-output* node and a *multi-input* node, respectively.

The transformation described above is implemented by the function `combine_farms`

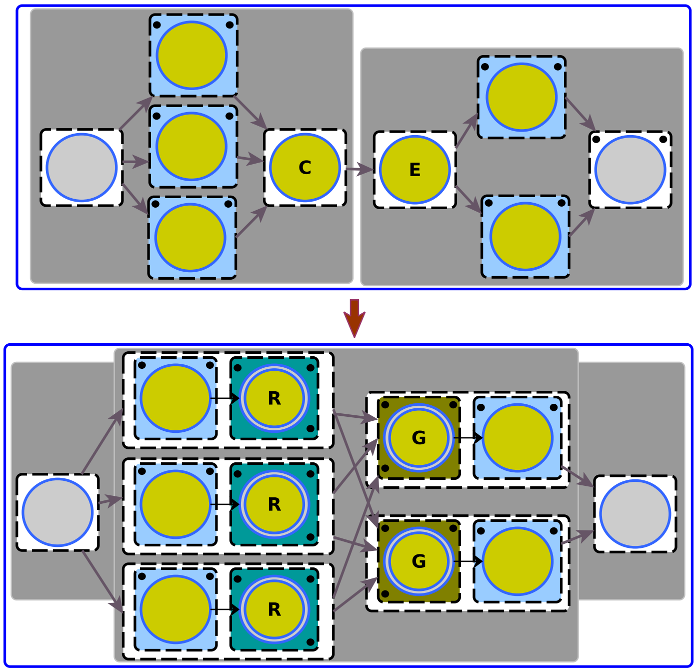


Figure 8-7: Combining two *farms* with user-defined Collector and Emitter.

whose signature is shown in the following snippet of code:

```
template<typename R_t, typename G_t>
const ff_pipeline combine_farms(ff_farm& farm1, const R_t *R,
                                ff_farm& farm2, const G_t *G,
                                bool merge);
```

The function's parameters R and G must be sequential FastFlow nodes, and, if not null, they will be replicated and combined through the *sequential node combiner* with the farm1 Workers and farm2 Workers, respectively.

The function returns a new *pipeline* building block that can be added to another pipeline or can be executed directly. Depending on the values provided as arguments (R, G and merge), the transformations executed by the combine_farms function are different. In particular, it provides four different transformations described in the following.

If merge is false we can distinguish the following cases:

- **Both R and G are valid pointers:** it produces exactly the topology sketched in Figure 8-7.

- **Only the `R` pointer is valid:** it produces a pipeline of a single *farm* whose Worker is an *all-to-all* building block where the nodes of L-Workers are a composition of `farm1`'s Workers and the node `R`.
- **Only the `G` pointer is valid:** it produces a pipeline of a single *farm* whose Worker is an *all-to-all* building block where the nodes of the R-Workers are a composition of the `G` node and `farm2`'s Workers.
- **Both pointers are not valid:** it produces the topology sketched in Figure 8-6.

. Instead, if the `merge` is `true`, the `combine_farms` function executes the following transformations:

- **Both `R` and `G` are valid pointers:** this produces a pipeline of two *farm*s where the first one does not have the Collector while the second one has as Emitter node the composition of the `R` node and `G` node (this is the case sketched in Figure 8-5 valid also for standard *farm* building blocks).
- **Only the `R` pointer is valid:** this produces a *pipeline* of two *farm*s where the first *farm* does not have the Collector node while the second *farm* has as Emitter the `R` node (this transformation substitutes the Emitter of the second *farm* and removes the Collector of the first *farm*, if present).
- **Only the `G` pointer is valid:** this is equivalent to the previous case (Only the `R` pointer is valid) where `R=G`.
- **Both pointers are not valid:** in this case, if the parallelism degree of the two *farm*s is the same, it applies the *farm fusion* transformation. If the parallelism degree of the two farms is different, the transformation applied is the same as the case when `merge=false`.

.

8.2.4 Utility functions

The *concurrency graph transformer* software layer provides a set of utility (or helper) functions to help the programmers to transform and modify building blocks. The *pipeline* building block provides methods to remove and add stages; the *farm* provides methods to change the Emitter node and Workers nodes; the `all-to-all` provides methods to change L- and R-Workers. These methods together with some utility functions that can be used to combine a sequential stage with a parallel building block enable a large set of possibilities for decomposing the building blocks and re-assembly them in a different way. Some of these combining functions are `combine_with_firststage` and `combine_with_laststage` which allow to combine a sequential node as first and last node of a pipeline whatever is the current first and last stage of the pipeline. Other similar functions are `combine_right_with_a2a` and `combine_left_with_a2a` for the *all-to-all* building block and `combine_right_with_farm` and `combine_left_with_farm` for the *farm* building block.

8.3 Evaluation

In this section, we evaluate the impact on the application performance of some of the transformations discussed in the previous sections. Specifically, we consider the *farm fusion* transformations in two distinct PARSEC benchmarks: *Ferret* and *Dedup*. Then we consider two different cases where the *all-to-all* building block is used to increase both resource efficiency and performance by modifying the RTS of two frameworks implemented on top of FastFlow: PiCo and Peafowl. Finally we will examine the `WindFlow` library by considering two use-cases. The `WindFlow` library provides the user with a set of Data Stream Processing parallel patterns implemented on top of the FastFlow *pipeline* and *farm* building blocks. It makes deep use of almost all transformations discussed in previous sections.

8.3.1 Farms fusion

To evaluate the *farm fusion* transformation we consider two applications from PARSEC benchmarks [47] (Princeton Application Repository for Shared-Memory Computers)[1], namely *Ferret* and *Dedup*. Their parallelization using parallel patterns has been studied in our previous work [129]. As reference platforms for the evaluation we consider the *Xeon*, *KNL* and *Power* systems described in Table. 6.1. We used as performance metric the *speedup* measured considering the so-called *region of interest* (ROI), which includes all parts sensitive to the parallelization.

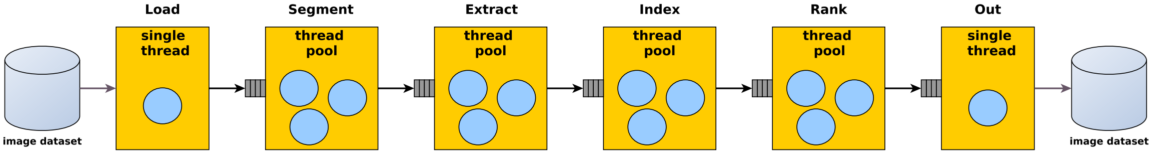


Figure 8-8: Pipeline schema of the *Ferret* application as implemented in the PARSEC benchmark using the PThreads library.

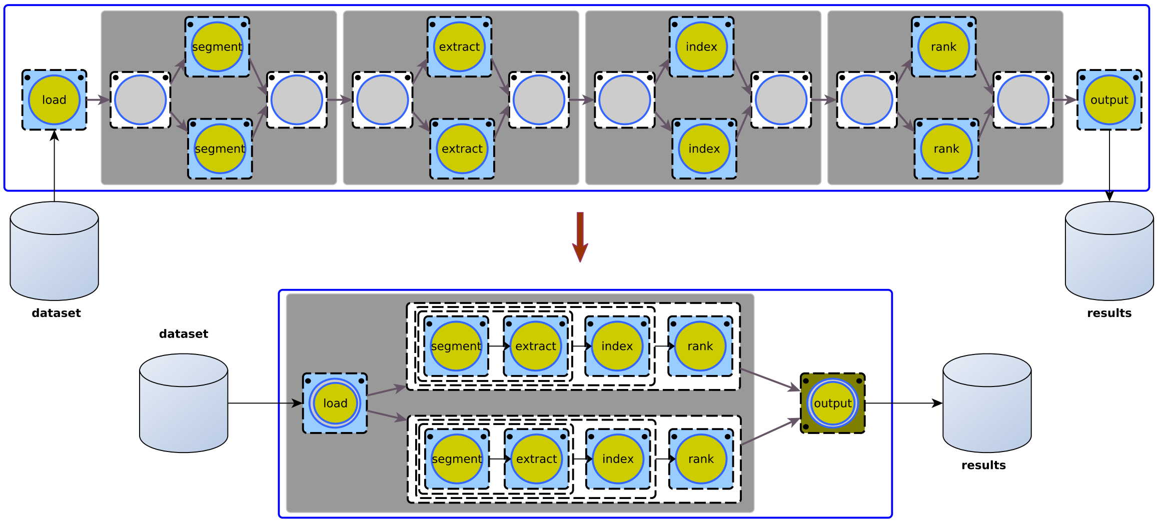


Figure 8-9: Building blocks implementation of the *Ferret* application. *Top)*: The same topology of the PARSEC PThreads version. *Bottom)*: Version obtained by calling `optimize_static` with `merge_farms`, `remove_collector` and `merge_with_emitter` optimization flags.

[1] We refer to the PARSEC version 3.0: `http://parsec.cs.princeton.edu/overview.htm`

Ferret. This is a stream-parallel benchmark. It is based on a toolkit used for content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes [228]. The toolkit is configured for image similarity search.

As sketched in Figure 8-8, this application can be modeled as a *pipeline* computation of six stages where the first and last ones are sequential components while the other four stages are internally concurrent. Communication channels between pipeline stages are implemented by using queues of fixed size. The FastFlow parallel version can be simply derived from the PARSEC PThreads version implementing the sequential stages using *sequential* building blocks and the parallel stages by using a *farm* building block [129]. The sequence of four *farm*s can be fused in a single *farm* because all of them have the same number of Workers and the same scheduling policy (the default round-robin policy). The Workers of the new *farm* building block are obtained by combining in the correct order the Workers of each *farm* in the pipeline. In addition the first and last stage (`load` and `output` nodes, respectively) can be combined with the Emitter and Collector of the new *farm*. The building block topology describing the *Ferret* application before and after the transformations is shown in Figure 8-9.

Code 23 shows the FastFlow parallel code. The business logic code of each stage is encapsulated in a sequential FastFlow building block. Then, each node is added to the `ff_Pipe` pattern (this is the high-level version of the *ff_pipeline* building block) respecting the pipeline order. The *farm* building block implementing the concurrent stages uses n replicas of the sequential building block implementing the stage. At line 26 all transformations specified with the optimization flags at line 25 are applied automatically to the pipeline `pipe`.

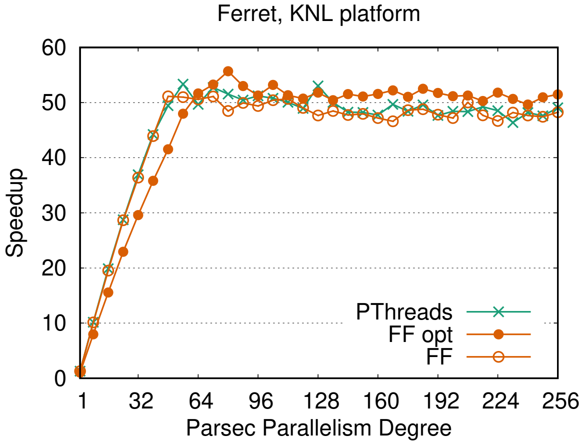
We measured the speedup of the *Ferret* application on all platforms. The plot in the left-hand side of Figure 8-10 shows the speedup on the *KNL* platforms. The performance improvement compared to the base FastFlow version is about 9% on the *KNL* platform and around 2% on the *Xeon* platform (see the table in the right-hand side of Figure 8-10). Basically, no performance improvement is obtained on the *Power* platform. The number of threads saved with the transformations is $3 \cdot (n + 2) + 2$

```
struct Load: ff_node_t<long,load_data> {  // first stage
    load_data *svc(long*) { <business logic code> };
} In;
struct Segment:ff_node_t<load_data,seg_data> { // second stage
    seg_data *svc(load_data *in) { <business-logic code> };
};
struct Extract:ff_node_t<seg_data,extr_data> { // third stage
    extr_data *svc(seq_data *in) { <business-logic code> };
};
struct Index:ff_node_t<extr_data,vec_query_data> { // fourth stage
    vec_query_data *svc(extr_data *in) { <business-logic code> };
};
struct Rank:ff_node_t<vec_query_data,rank_data> { // fifth stage
    rank_data *svc(vec_query_data *in) { <business-logic code>};
};
struct Output:ff_node_t<rank_data> { // sixth stage
    void *svc(rank_data *in) { <busines-logic code> };
} Out;
auto farm1 = ...; auto farm2 = ...; // creating farms stages
auto farm3 = ...; auto farm4 = ...;
ff_Pipe<> pipe(In, farm1, farm2, farm3, farm4, Out);
OptLevel opt;
opt.remove_collector  =true;
opt.merge_with_emitter=true;
opt.merge_farms       =true;
optimize_static(pipe, opt);
pipe.run_and_wait_end(); // pipeline execution
```

Code 23: FastFlow code of the *Ferret* application.

where n is the parallelism degree of the *farm* building block.

One of the main limitations for improving the performance of the *Ferret* application is the loading of input files in the main memory. Since input files can be computed in parallel and the order of which one is computed first is not important, the part of the *load* module that reads files into main memory can be parallelized by moving it into the farm building block (see Figure 8-11, we divided the *load* module into two modules *load1* and *load2* the first opening the file and the second loading its content into main memory, respectively). The speedup of the modified version is reported in Figure 8-12. As can be seen, the improvement is considerable both on the *KNL* (about 66%) and *Power* (about 38%) platforms. This is an interesting case



	max. speedup		
	PThreads	**FF**	**FF opt**
Xeon	25.8	25.4	25.9
KNL	52.7	51.1	55.7
Power	27.4	25.3	25.4

Figure 8-10: *Left)*: Speedup of the *Ferret* benchmark on the KNL platform. *Right)*: Maximum speedup obtained by the PThreads, FastFlow (FF) and the FastFlow optimized (FF opt) versions on the Xeon, KNL and Power platforms.

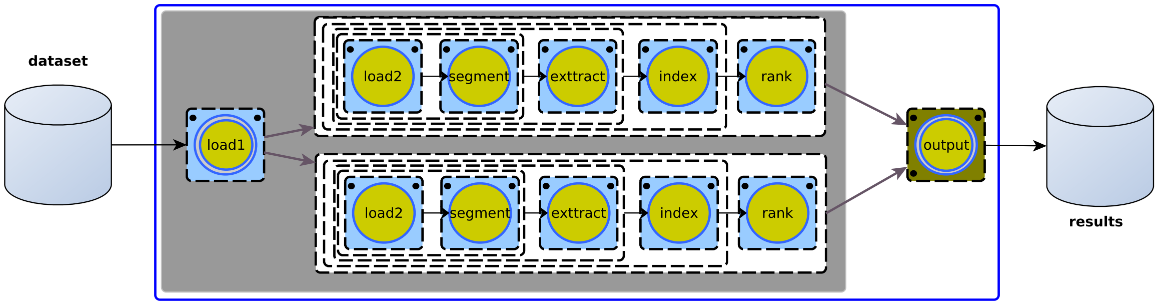
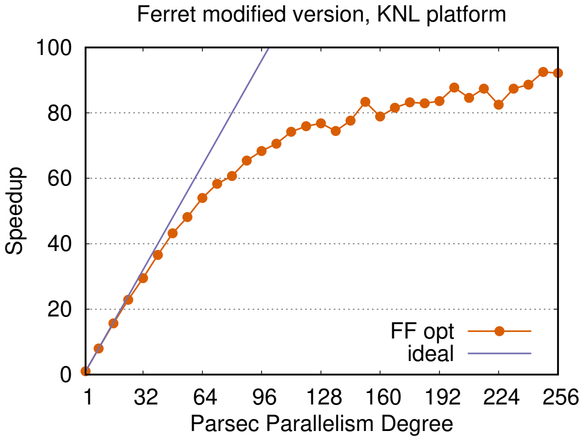


Figure 8-11: Modified FastFlow version of the *Ferret* application.



	max. speedup
	FF opt
Xeon	25.8
KNL	92.5
Power	35.2

Figure 8-12: *Left)*: Speedup of the modified version of the *Ferret* application on the KNL platform. *Right)*: Maximum speedup obtained by the modified and optimized FastFlow version on the Xeon, KNL and Power platforms.

where pattern composition allows the programmer to prototype alternative versions that are more efficient than the initial one, by changing just a few lines of code.

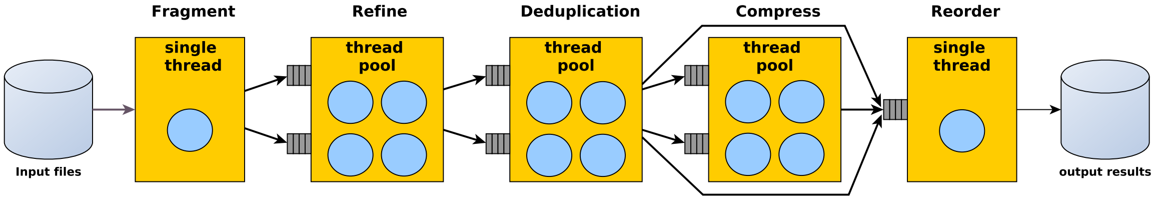
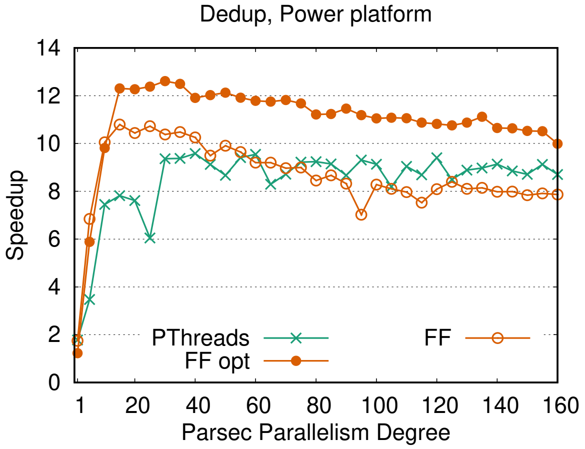


Figure 8-13: Pipeline schema of the *Dedup* application as implemented in the PARSEC benchmark using the PThreads library.

Dedup. Like *Ferret*, *Dedup* is a coarse-grained streaming application that compresses a data stream with a combination of global and local compression phases called "deduplication". The PThreads version implements a pipeline with five stages, where each middle stage is implemented with a thread pool (the first and last stages are single-threaded). Figure 8-13 shows a representation of the *Dedup* pipeline. The PThreads implementation is not a "pure" pipeline: results produced by the third stage may be transmitted directly to the last stage skipping the fourth stage. Furthermore, the second stage can generate more output items for each input data element. To lower the contention on communication channels, the PThreads implementation uses multiple queues of fixed size to connect consecutive stages. Each queue is assigned to a subset of threads in the same pool.

The first stage (*Fragment*) reads the data stream from the disk and then partitions the data at fixed positions; then, it produces in output a stream of data chunks. Each chunk can be processed independently from the other chunks. The second stage (*Refine*) further partitions the input chunk into smaller fine-grained chunks generating a nested stream. The third stage (*Deduplication*) checks if the chunk has already been compressed in the past by accessing a hash table. If so, the chunk is marked as *duplicate*. The fourth stage (*Compress*) compresses all the chunks that are not marked as *duplicate*, and updates the corresponding table entries. To ensure correctness in the access to the table performed by the *Deduplication* and the *Compress* stages, each bucket in the hash table is protected with a *lock*. Finally, the *Reorder* stage writes the final compressed output data into the output file. If the input chunk was marked as *duplicate*, it stores a "reference" to the corresponding chunk. This stage reorders the data chunks as they arrive to match the original order of the uncompressed data.



	max. speedup		
	PThreads	**FF**	**FF opt**
Xeon	6.7	9.2	8.8
KNL	5.8	6.2	6.3
Power	9.6	10.8	12.6

Figure 8-14: *Left)*: Speedup of the *Dedup* benchmark on the *Power* platform. *Right)*: Maximum speedup obtained by the PThreads, FastFlow (FF) and the FastFlow optimized (FF opt) versions on the Xeon, KNL and Power platforms.

This stage represents the main bottleneck of the *Dedup* pipeline, both due to data reordering and to I/O operations.

The *Dedup* application can be modeled as a FastFlow pipeline, where the first stage and the last stage are sequential nodes, while the three middle stages are instances of the *farm* building blocks. We implement the bypassing mechanism between the third and fifth stage by adding a flag to each data element. The flag is set to `true` if the data element must be transmitted directly to the last stage. In that case, the *Compress* stage only forwards the element to the final stage without any further processing. The building block schema of this parallelization is very similar to the *Ferret* one (top part of Figure 8-9). The middle *farm* can be automatically merged obtaining a single *farm* building block like the one obtained in the *Ferret* applications (bottom part of Figure 8-9).

The speedup results are shown in Figure 8-14. The plot on the left-hand side shows the speedup figures obtained in the *Power* platform. In this case the performance improvement is about 16% on the *Power* platform and around 1% on the *KNL* platform. Strangely in the *Xeon* platform the building blocks fusion does introduce a small performance degradation.

As for *Ferret*, also for this benchmark, it is possible to derive a new version of the *Dedup* application. The new version exploits the ordering feature of the *farm* building

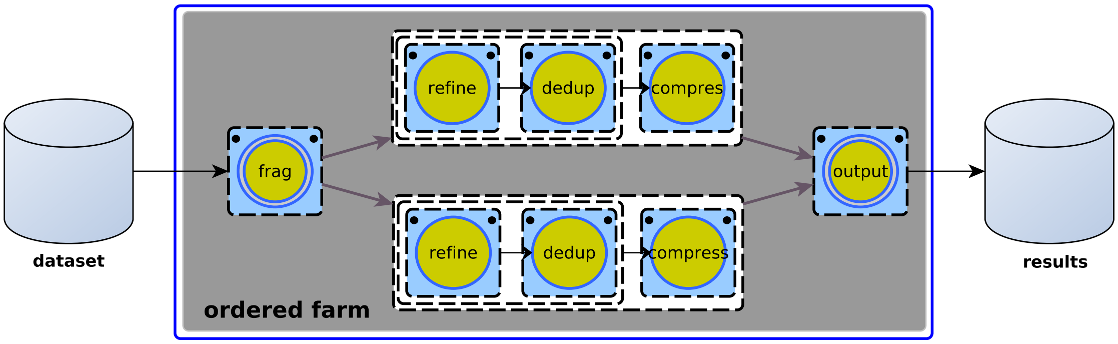
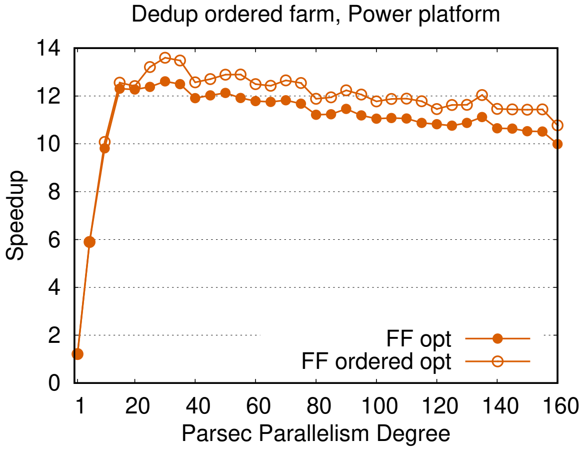


Figure 8-15: Modified FastFlow version of the *Dedup* application.



	max. speedup **FF opt**
Xeon	9.3
KNL	6.6
Power	13.6

Figure 8-16: *Left)*: Speedup of the modified version of the *Dedup* application on the Power platform. *Right)*: Maximum speedup obtained by the modified and optimized FastFlow version on the Xeon, KNL and Power platforms.

block, allowing to lighten the computational burden to the last stage of the pipeline that does not need to keep the ordering of chunks. The last stage (*Reorder*) has been substituted by the *Output* stage that simply writes the already ordered results into the disk. The ordering policy used in the experiments is the *round-robin* one. The building block structure of the new version is sketched in Figure 8-15.

The speedup figures obtained on the *Power* platform for the new version are shown in the left-hand side of Figure 8-16. The new version provides an 8% improvement on this platform and a marginal improvement on the *KNL* and *Xeon* platforms (see the table on the right-hand side of Figure 8-16). For all platforms the advantage compared to the original PThreads implementation is significant from about 14% to about 40%.

All in all, the *farm fusion* transformation allows us to increase the performance in almost all cases tested. The reduction of the number of nodes used in the FastFlow RTS is significant, and the resulting topology uses a lower number of threads with respect to the initial versions. Moreover, we have shown that the building blocks composition approach allows the programmer to quickly prototype different versions of the same application by applying modifications to specific parts of the code. Both for *Ferret* and *Dedup* applications, the versions obtained changing just a few lines of code and re-using all other already written parts, combined with the process network optimization for reducing the number of threads, allowed us to obtain significant advantages in terms of speedup in all three platforms considered. Despite the fact that different versions could also be implemented in PThreads (or other programming models), this would require re-implementing from scratch all the communications and the data dependencies between different parts of the parallel application. On the contrary, in pattern-based model dependencies and communications are implicitly coded in the pattern.

8.3.2 All-to-All Introduction

In this section we evaluate the transformation that introduces the *all-to-all* building block as merging of two *farm*s in pipeline (see Section 8.2.3). To this end, we consider two different application scenarios: 1) the Word Count application implemented in the PiCo parallel framework [251]; 2) a fine-grained network application (Protocol Identification) implemented using the Peafowl framework [110]. The aim is to evaluate the performance of the proposed graph transformations in two real use-case scenarios.

The PiCo framework. *PiCo* (Pipeline Composition), is a C++ framework aiming at enhancing the performance of Big Data Analytics applications on multi-cores through a unified model for batch and stream processing. It provides an expressive programming model supported by a functional abstract semantics. The abstract model is coupled with a concrete API expressed using modern C++, thus ensuring good code portability. One distinguishing feature of PiCo is the polymorphic

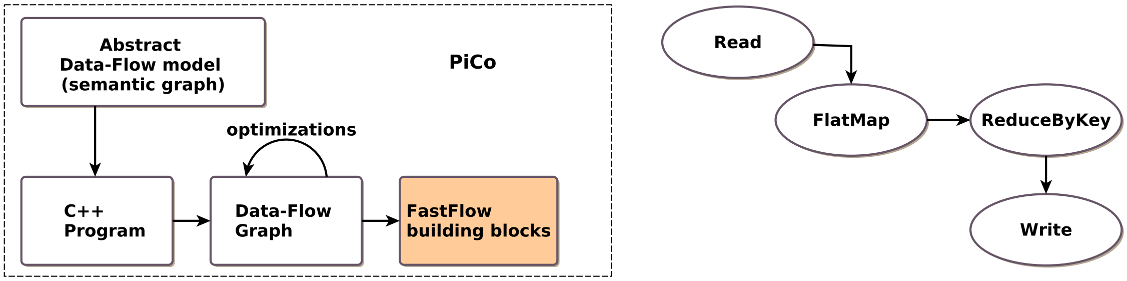


Figure 8-17: *Left)*: *PiCo* logical implementation schema. *Right)*: Word Count *PiCo* semantic graph.

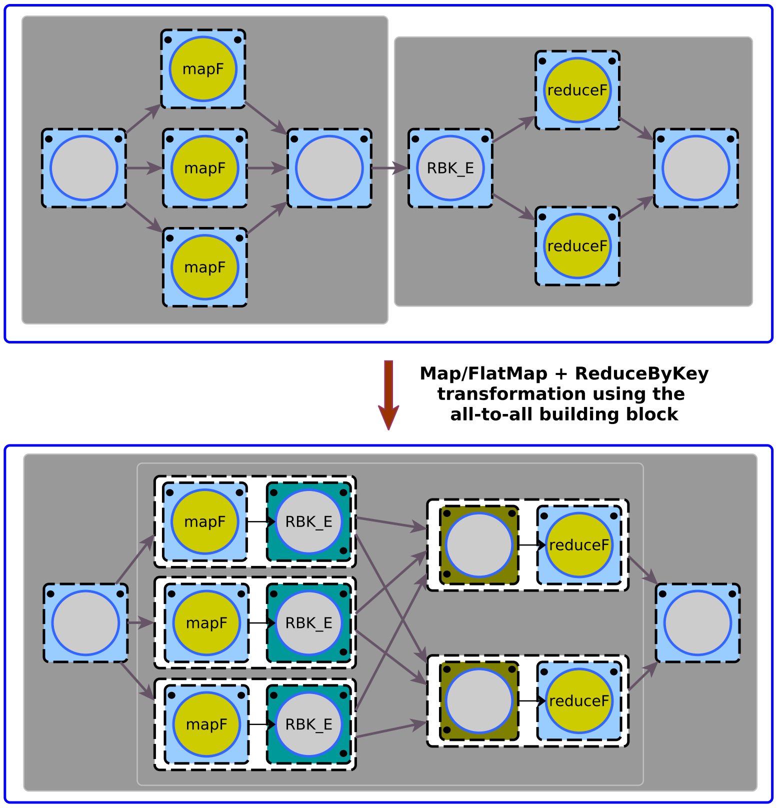


Figure 8-18: Building blocks implementation of the *FlatMap+ReduceByKey* optimization in PiCo.

pipelines, that allow uniform programming for both stream and batch processing.

PiCo is currently implemented on top of FastFlow and its logical architecture is sketched in the left-hand side of Figure 8-17. It also provides a set of rewriting rules for optimizing the Data-Flow graph describing the application. For instance, in a *Map+ReduceByKey* composition, part of the reducing phase is moved into the *Map* part to be computed in parallel.

It offers a set of *operators* as core components of *Pipeline* (e.g., *Map*, *FlatMap*, *ReduceByKey*). These operators are implemented on top of FastFlow *pipeline*, *farm* and *sequential node* building blocks.

We have modified the *PiCo* optimization layer to introduce the *all-to-all* building block each time the Data-Flow graph contains a sequence of *Map/FlatMap + ReduceByKey* operators. The building block implementation of the *FlatMap+ReduceByKey* optimization using the *all-to-all* building block is shown in Figure 8-18.

The Emitter of the *ReduceByKey farm* (*RBK_E* in the figure) is replicated and combined with each *Map* Worker. In addition, in front of each *ReduceByKey* Worker a service *multi-input* node is combined with each Worker to set the proper input cardinality of the R-Workers of the *all-to-all* building block. All these operations are automatically implemented by the `combine_farms` function (see Section 8.2.3).

Word Count in PiCo. Although the "Word Count" is a very simple application, in the context of Big Data analytics it is considered a reference test case. It finds the number of occurrences of each word from an input text file. The input is read line by line from the input file. Each line is tokenized into a sequence of words using a *Map* pattern (*FlatMap* in PiCo) as each line contains a varying number of words. Then, each word w is mapped to a key-value pair $< w, 1 >$ which are grouped by-word and then the values (i.e. the 1s) are summed up by a *ReduceByKey* pattern node. Finally, the result (i.e. one pair per word along with its number of occurrences in the text), is written into the text file. The semantic graph of the Word Count application in PiCo is sketched in the right-hand side of Figure 8-17.

Figure 8-19 shows the results obtained testing the Word Count application on the *Xeon* platform with a large number of keys (test called *many keys*) and a small number of keys (test called *the few keys*). The many keys test uses about 350K keys while the few keys test uses 10K keys. The input text file has two hundred thousand lines and a size of about 1.3GB. For the many keys test, we fixed the number of Workers of the *FlatMap* to 24 and we varied the parallelism degree of the *ReduceByKey* component. In the left-hand side of Figure 8-19 is shown the optimized version (FF opt) obtains a

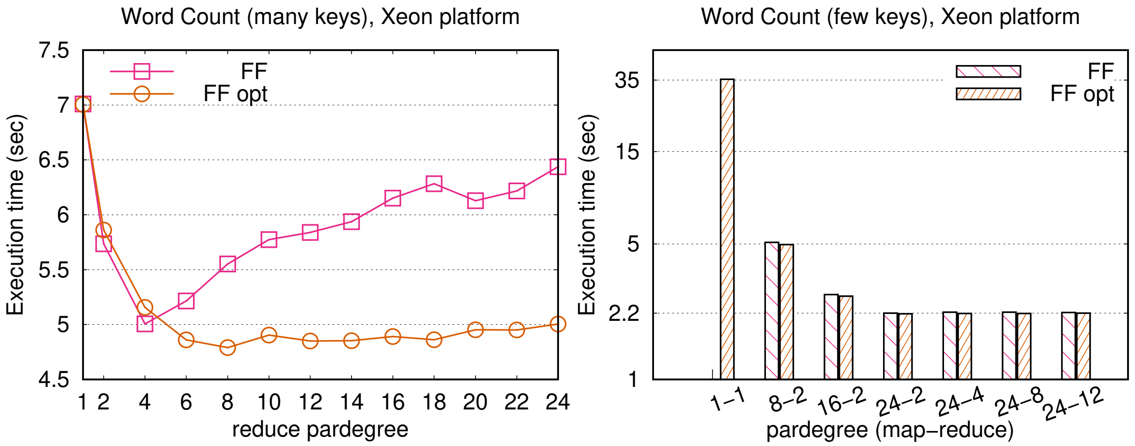


Figure 8-19: *Left)*: Word Count execution time in the configuration "many keys". The *FlatMap* uses 24 Workers whereas the number of Workers of the *ReduceByKey* varies. *Right)*: Execution time for "the few keys" configuration. The x-axis reports the number of Workers of the *FlatMap* and *ReduceByKey* used.

better execution time (the improvement is about 5%). However, the most interesting aspect is that the transformed version does not degrade considerably if the number of Workers of the *ReduceByKey* increases thus demonstrating a more stable behavior compared to the non-optimized FastFlow version.

The test with the few keys is reported in the right-hand side of Figure 8-19. In this case, since the *ReduceByKey* part is less critical, we keep its number of Workers fixed (equal to 2) varying the number of Workers of the *FlatMap* component up to the available number of cores. Then, we start increasing that number of the *ReduceByKey* Workers to see if the performance remains stable. As can be seen from the figure, there are almost no differences between the default and the optimized FastFlow version and when the parallelism degree of the reduce part increases the performance does not decrease as expected.

This simple tests demonstrate that *all-to-all* transformations are beneficial for optimizing the *Map+Reduce* compositions. It provides higher scalability when the *Reduce* part may be a bottleneck and does not introduce extra overheads when the *Reduce* part is not critical.

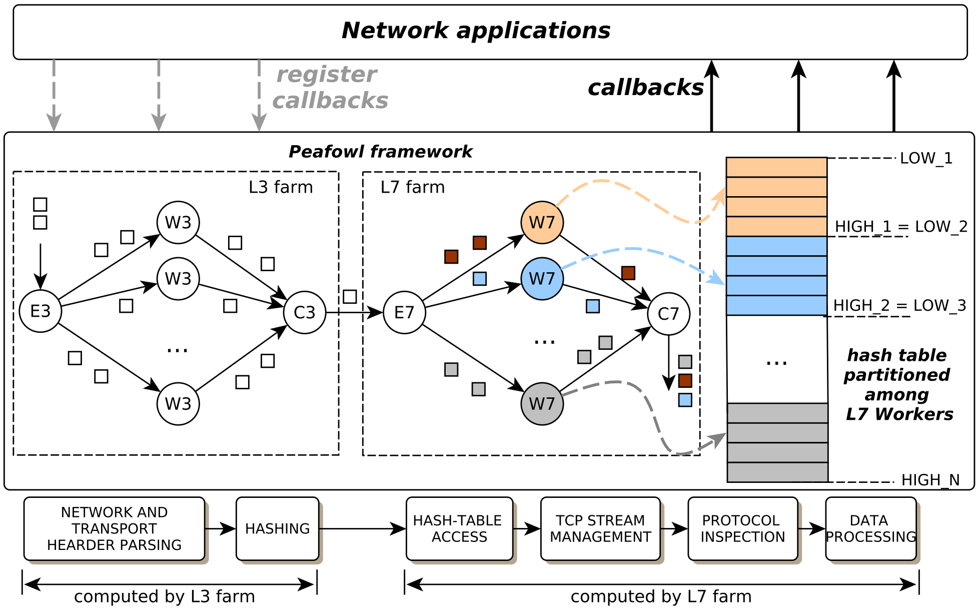


Figure 8-20: *Peafowl* parallel framework logical architecture.

The Peafowl framework

Peafowl is a flexible and extensible Deep Packet Inspection (DPI) framework which can be used to identify the application protocols carried by IP (IPv4 and IPv6) packets and to extract and also process data and metadata carried by those protocols[2]. The framework is structured using the *Task-Farm* pattern whose Workers access a carefully designed network flows table that is partitioned among all Workers. It is currently implemented by using the FastFlow building blocks *pipeline* and the *farm* [110]. Each Worker processes only the network flows belonging to its flow table partition. Therefore, the *farm* Emitter uses ad-hoc scheduling function for the input packet by hashing the information contained in the packet header. Once a generic Worker receives the packet from the Emitter, by using the state information contained in the hash-table, it manages the TCP stream, infers the protocol, and if required, after the protocol has been identified, it executes further processing by calling proper user's application callbacks.

However, in the case of high-input rate, the time required to parse the network and transport headers and to apply the hash function might be significant and therefore

[2] Peafowl project home: `https://github.com/DanieleDeSensi/peafowl`

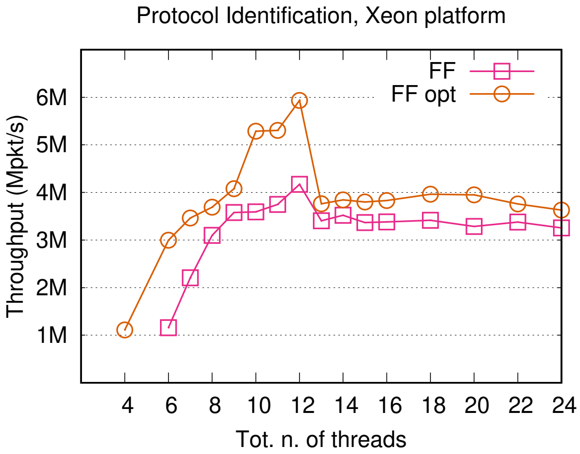
the centralized *farm* Emitter may limit the framework scalability. To avoid this sequential bottleneck, the *Peafowl* framework allows the user to replace the *farm* Emitter with a second *farm* (called L3 farm) where the Emitter is replicated. The resulting topology is a two-stage pipeline where the first and second stages (L3 farm and L7 farm, respectively) are two *farm*s building blocks (see Figure 8-20).

In the tests, we considered the *Peafowl* configuration with two *farms*. Specifically, we have modified the framework to introduce the *all-to-all* building block for merging them. The L3 Workers of the first L3 *farm* are the L-Workers of the *all-to-all* while the L7 Workers of the L7 *farm* are the R-Workers of the *all-to-all* building block. This transformation, that allows us to spare two FastFlow threads, one for the L3 Collector and one for the L7 Emitter, can be introduced because the L7 Emitter does not have any internal state and can be replicated. The resulting building block topology is very similar to the one obtained for the Word Count test in the *PiCo* framework (see the bottom schema in Figure 8-18). The R-Workers of the *all-to-all* use an *on-demand* scheduling policy to dispatch packets toward the L-Workers (implemented by a *multi-input* node combined with the L7 Worker). The L7 Worker for each HTTP packet identified, calls a user's callback that simply inspect the packet payload. The resulting computation per packet executed by the L7 Worker is very fine-grained with an average execution time less than $1\mu s$ on the *Xeon* platform.

We tested the network application with a synthetic dataset containing several millions of packets, most of them being HTTP packets (about 90%). Packets are directed to the *Peafowl farm* at maximum speed by a dedicated node part of the application pipeline. This test aims to measure the maximum throughput sustained by the system for the two configurations considered.

The results obtained varying the number of Worker threads and considering the default *Peafowl* configuration (i.e. two *farms* in *pipeline* – FF) and the transformed configuration (i.e. one *farm* with a single *all-to-all* Worker – FF opt) are reported in Figure 8-21.

The two configurations have been compared with an equal number of RTS threads. The default version uses ($L3Workers+L7Workers+4$) threads while the transformed



threads	**config.** **FF,FF opt**	**FF**	**FF opt**
4	-, 1-1	-	1.2M
6	1-1, 1-3	1.2M	3.0M
8	1-3, 2-4	3.1M	3.7M
10	1-5, 2-6	3.6M	5.3M
12	1-7, 2-8	4.2M	5.9M
14	4-6, 6-6	3.5M	3.8M

Figure 8-21: *Left)*: Throughput (millions of packets per second – Mpkts/s) on the *Xeon* platform of the *Peafowl* Protocol Identification application with and without the *all-to-all* transformation. *Right)*: Table showing the configuration of L3 and L7 Workers used for the given number of RTS threads. For instance for the case 10 threads, the FF version uses 1 L3 Worker and 5 L7 Workers, whereas the FF opt version uses 2 L3 Workers and 6 L7 Workers.

version uses $(L3Workers + L7Workers + 2)$ threads. As can be seen, the FF opt version is able to obtain higher-throughput. This is because the transformed version can use two extra Worker threads for the L3 Worker set and/or L7 Worker set. The exact number of Workers used for the R-Workers and L-Workers is shown in the table at the right-hand side of Figure 8-21. For example, for the case with 12 threads, the FF version uses 1 L3 Worker and 7 L7 Workers, whereas the FF opt version uses 2 L3 Workers and 8 L7 Workers.

It is worth noting that, given the very fine-grained computation, the highest throughput is obtained with 12 threads in both cases. This is because for that configuration all threads are packed in a single CPU of the *Xeon* platform, and all threads share the last-level chip cache. The FF opt version is more efficient because the two spared service threads are used to do useful work (one extra Worker both for the L3 and L7 parts), and this leads to an absolute gain of more than 40%. When increasing the number of threads, some of them are pinned onto cores of the second CPU, thus not sharing any cache and with a higher cost to dispatch packets from L-Workers and R-Workers (for the higher cost of inter-thread communications not sharing chip cache, see Chapter 6).

To conclude, we would like to remark that both *PiCo* and *Peafowl* frameworks that we used in our tests were already optimized with excellent performance figures in the application considered. The extra boost in the performance we were able to obtain implementing suitable graph transformations by using the *all-to-all* building block, confirms that the new building block widens the opportunity to introduce optimizations in addition to increasing composability.

8.3.3 Multiple transformations

Here we consider the WindFlow parallel library targeting Data Stream Processing (DSP) applications. It is currently under development at the Parallel Computing Group of the Computer Science Department of the University of Pisa[3]. First, we provide a brief introduction of the library and then we discuss the optimizations currently employed by the library which make extensive use of the FastFlow graph transformations offered by the new FastFlow transformations software layer.

The WindFlow library. The library provides the user with a set of ready-to-use, user-friendly implementations of some of the most relevant patterns in the DSP context for multi-cores platforms [127, 246, 248]. One of the main characteristics of the library is the possibility to exploit pattern nesting to improve application performance and enlarge the design space in DSP computations. The WindFlow library, implemented on top of the FastFlow library, makes large use of the transformations presented in this chapter with the aim to both reduce the number of concurrent entities (nesting of patterns introduce many “service” nodes) and to remove potential bottlenecks in the process network topology for high input rate.

Sliding windows model. Many applications (e.g., stock market prediction, web-mining), do not consider outdated elements relevant to make near future decisions. This peculiarity gives rise to the *windowing paradigm* where continuous queries are applied to the most recent part of the data stream by using *sliding windows* buffers [31, 164]. How large is this part depends on the *window size* and on the *sliding factor*

[3] Parallel Programming Models Group: http://calvados.di.unipi.it/paragroup/

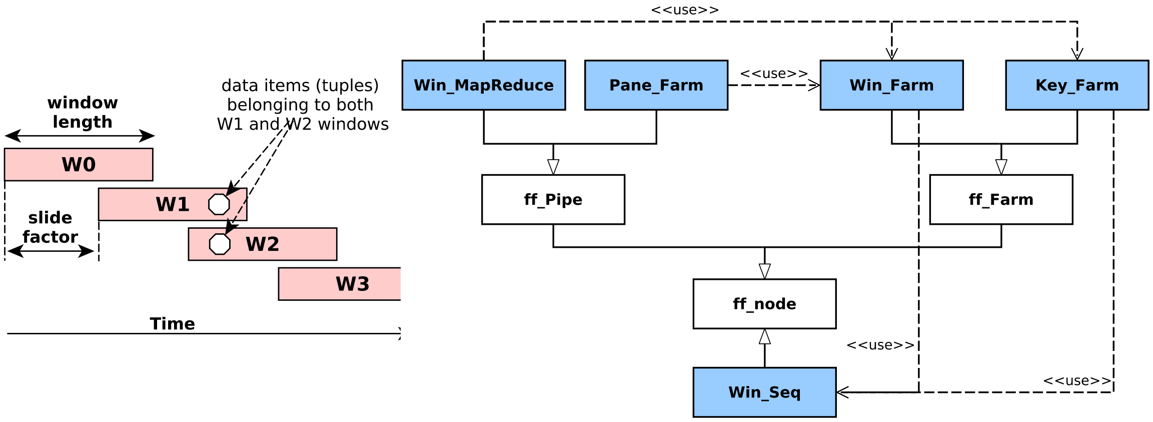


Figure 8-22: *Left)*: Sliding window model of computation. *Right)*: UML diagram of the `WindFlow` classes and relationship with the FastFlow classes.

(see the left-hand side of Figure 8-22). The choice of the sliding factor and the window length, leads to several configurations that differ by the existence of a possible overlapping between consecutive windows. In the most simple and general case, a number of stream items (also called *tuples*) belong to multiple consecutive windows thus allowing significant *data reuse*.

WindFlow patterns. The WindFlow library currently offers five streaming patterns, one with sequential semantics and four parallel patterns all implemented as C++ template classes extending the FastFlow classes. Figure 8-22 depicts the organization of the library's source code. The sequential pattern named `Win_Seq` represents the basic building block that directly extends the FastFlow sequential node (through the `ff_node` class). The parallel patterns, `Win_Farm`, `Key_Farm`, `Pane_Farm` and `Win_MapReduce`, are based on either the *pipeline* or *farm* building blocks extending the FastFlow `ff_Pipe` or the `ff_Farm` classes. The *farm* building block is configured with customized distribution and collection policies designed to enforce the pattern-specific parallel semantics. The patterns enable the programmer to exploit different level of parallelism: *inter-key parallelism* where replicas of the same transformation stage process items of different key groups in parallel; *inter-window parallelism* where successive windows of the same sub-stream could be assigned and processed in parallel by distinct replicas; and *intra-window parallelism* when specific properties of the query function are known (e.g., associativity and commutativity), the processing of each window can be executed in parallel following the *map-reduce* pattern, by splitting the window content and aggregating the partial results into a window-wise result. Although some patterns already exploit more than one dimension of parallelism, pattern nesting makes it possible to combine them in a general way within a single application structure to have the possibility to explore many different alternative solutions. In the following, we describe in detail only the patterns used in our tests. The library supports both *count-based* and *time-based* windows in different configurations. In the first case, the window stores a fixed number of elements (i.e. the window size), in the second case the number of elements in the window depends on the input data rate. The `Win_Farm` pattern [127] supports inter-window

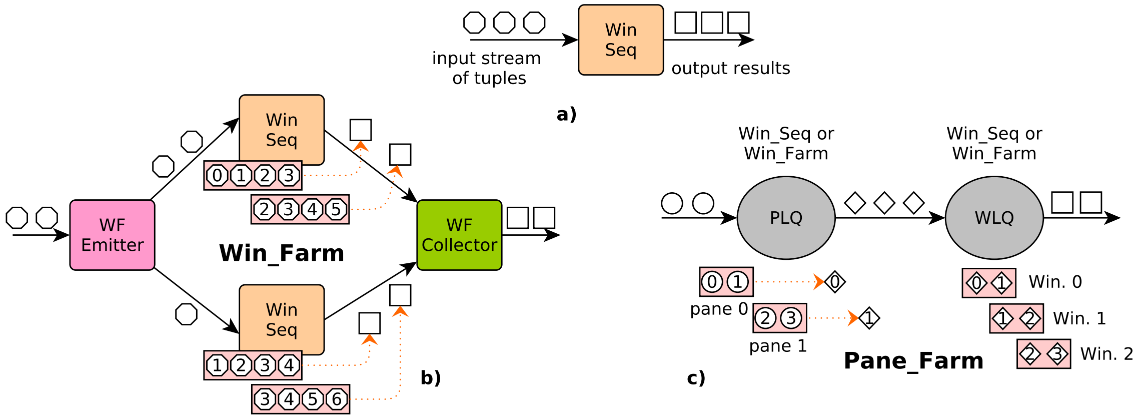


Figure 8-23: Logical schema of some `WindFlow` patterns: a) `Win_Seq`; b) `Win_Farm`; c) `Pane_Farm`.

parallelism by extending the basic *farm* pattern of FastFlow by using a specialized Emitter node for the scheduling policy. It replicates an inner pattern multiple times according to the *parallelism degree* chosen by the user. The internal instances work in parallel on distinct windows, regardless of their belonging to the same or different sub-streams. The inner pattern can be any pattern in the library: notably, `Pane_Farm` and `Win_MapReduce` patterns are natural candidates to be used within a `Win_Farm`, in addition to `Win_Seq` instances that is the default case (in Figure 8-23(b) is shown a `Win_Farm` containing as a nesting pattern the `Win_Seq`). Concerning the scheduling policy, each input item is distributed to all the internal instances that are in charge of computing at least one window where that item is included. This implies a *multi-*

cast send. The Win_Farm pattern has a custom Collector node which produces results with the same key attribute in increasing order of identifier by implementing a proper buffering of out-of-order results.

The Pane_Farm models a parallel implementation of the *paned approach* to continuous queries [224, 246]. The pattern supports inter-window parallelism, by executing parts of different windows in parallel, and intra-window parallelism by using some partial results of previous windows to avoid recomputing a new window from scratch. To this end, it uses a two-level aggregation based on the notion of *pane*: panes are windows of length equal to $p = GCD(w, s)$ (where w and s are the window size and the slide size, respectively). The computation consists in two phases: the *Pane-Level sub-Query* (PLQ) computes a partial result for each pane, while in the *Window-Level sub-Query* (WLQ) the results of the w/p panes that belong to the same window are merged to produce the corresponding window result. The fundamental property is that the partial results of panes shared among different windows are reused by saving computation time. The Pane_Farm pattern is a pipeline of two stages, possibly parallel inside, see Figure 8-23(c). The first stage is in charge of computing the intermediate results of the panes, while the second stage assembles the pane results to build the results of each window. The pattern is implemented as a class extending the ff_pipe of the FastFlow library. The pipeline is built with two stages that can be instances either of the sequential pattern or the Win_Farm pattern according to the concurrency level chosen by the user for that stage. The first stage works with windows with length and slide equal to p (the pane length). The second stage always works with count-based windows of length w/p and slide s/p.

Building block optimizations. The transformations currently used by the WindFlow library for the Pane_Farm pattern are those sketched in Figure 8-24, one for each possible implementation of the pattern. Specifically, the transformation with the label *d)*, requires to introduce a new node (called *ORD* in the figure) to keep the ordering of the pane results produced in parallel by the PLQ Workers. These nodes are automatically added by the library when the transformation is used.

The Pane_Farm pattern is an example of an advanced and complex pattern im-

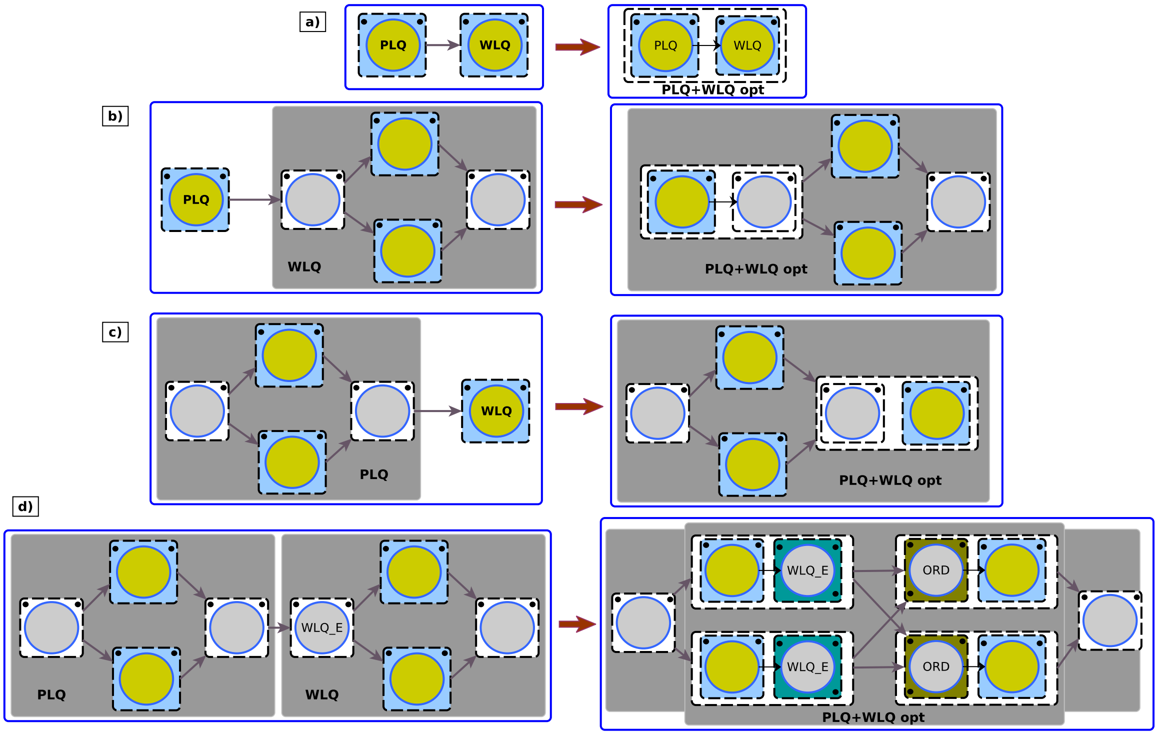


Figure 8-24: `Pane_Farm` pattern implementations and their associated transformations. a) Both stages are sequential; b) First stage sequential and the second stage parallel (instance of `Win_Farm`). c) First stage parallel (`Win_Farm`-based) and second stage sequential; c) Both stages are parallel.

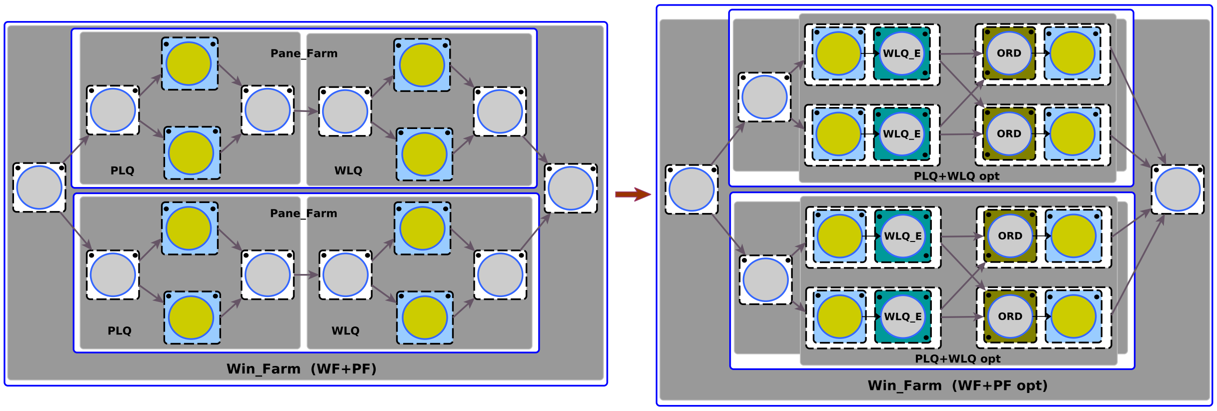


Figure 8-25: Building block schema of the “WF+PF” pattern and its transformed version (`WindFlow` `opt2` optimization level).

plemented as combination of simpler *farm* building blocks properly configured with input parameters and proper scheduling and gathering policies.

Figure 8-25 shows the `Win_Farm` + `Pane_Farm` pattern composition (i.e. a `Win_Farm` whose Workers are `Pane_Farms`) and the supported transformation. This pattern composition (called simply "WF+PF" for brevity) allows us to combine the transformations of the `Pane_Farm` pattern (see Figure 8-24) with the possibility to remove the Collectors of nested PF *farm*s into another *farm* (see Section 8.2.1).

To enable the `WindFlow` transformations (not only the ones presented here), the library offers two command-line options: `opt1` and `opt2`. The first option enables the optimizations *a)*, *b)* and *c)* sketched in Figure 8-24, while the `opt2` options enables all transformations including *d)* in Figure 8-24 and the one in Figure 8-25.

Evaluation

To evaluate the impact of the *building block* transformations presented in the previous sections for the `WindFlow` library, we consider one synthetic benchmark and a simple application. Both of them use the `Pane_Farm` and `Win_Farm` patterns described before. The test is structured as a three-stage pipeline, where the first stage is the tuples generator, the middle stage is the pattern selected, and the last stage is the results collector.

We first evaluate the `Pane_Farm` pattern and in particular the optimization of combining both the PLQ and WLQ *farm*s using the *all-to-all* building block (see Section 8.2.3). We consider a synthetic use-case where the functions computed in the PLQ and WLQ Workers introduce a negligible delay. This is an extreme case that allows us to analyze the case where the service nodes used to connect the two *farm*s implementing the pattern may introduce a bottleneck in the `Pane_Farm` pattern.

The synthetic `Pane_Farm` benchmark considers count-based windows of size $s = 1000$ and a sliding factor of $s = 10$ tuples. The objective is to analyze the maximum throughput sustained by the parallel implementation varying the number of Workers used in the PLQ and WLQ farms. With an input rate of about $R = 700,000$K data elements per second (700Ktuple/s) the maximum number of results per second produced by the pattern is $(R \cdot s)/w)$, that is $7,000$ results/s for the configuration considered. A lower number of results means that the `Pane_Farm` pattern is a bottleneck, i.e. one of the two farms is not able to sustain the data rate.

The results obtained on the *KNL* platform are shown in the left-hand side of

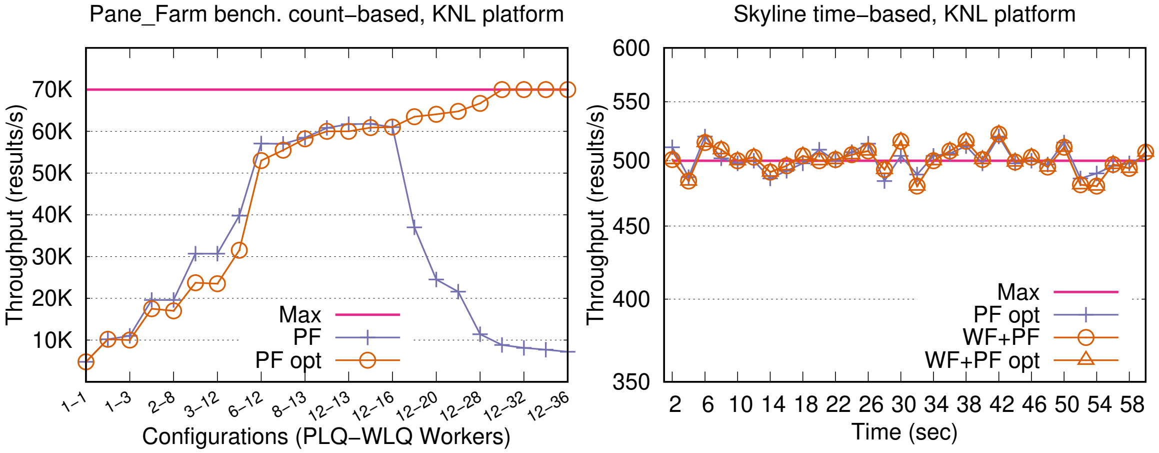
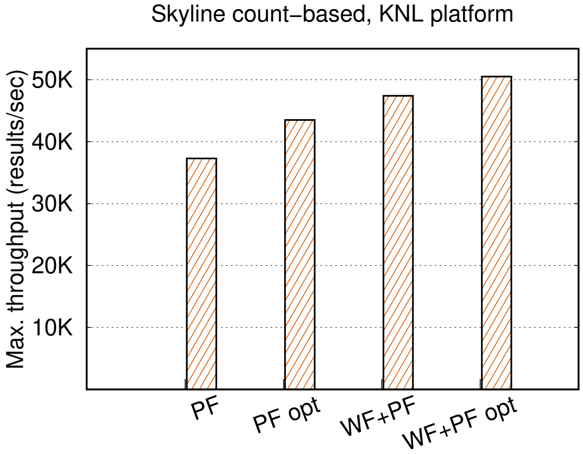


Figure 8-26: *Left)*: Pane_Farm benchmark, count-based configuration, rate 700 Ktuple/s, $w = 1000$s $s = 10$. *Right)*: Skyline test, time-based configuration, rate about 500K tuple/s, $w = 1$s $s = 2$ms.

Figure 8-26. On the x-axis is reported the number of Workers used for the PLQ and WLQ *farms*, respectively. It is possible to observe that the default version (*PF*) is not able to reach the maximum output rate for all configurations tested. Instead the transformed version (*PF opt* in the figure) reaches the maximum value with 12 Workers for the PLQ *farm* and starting from 30 Workers for the WLQ *farm*. This is due to the parallelization of the Emitter node of the WLQ *farm* that introduces a sequential bottleneck in the default configuration when the number of WLQ Workers is high. This is confirmed by the shape of the throughput curve of the default version. The maximum value is reached with 12 PLQ Workers but as soon as the number of WLQ Workers increases above 16 the throughput significantly degrades.

The second test considers a time-base configuration and a real kernel: the Skyline. The Skyline query computes the set of non-comparable tuples using the Pareto dominance. The operation is associative and commutative and therefore can be efficiently computed in streaming using an incremental approach. However, WindFlow already provides a batch-based implementation leveraging the SkyBench C++ library which implements the efficient algorithm proposed by Lee & Wang [221]. We analyzed both the Pane_Farm and the Pane_Farm nested into a Win_Farm patterns (i.e. “WF+PF”). For both versions we considered the corresponding transformed version obtained by using the command-line optimization flag opt2.



	#Workers (PLQ-WLQ)	max th.
PF	(6-52)	37.3K
PF opt	(7-53)	43.5K

	#Workers WF-(PLQ-WLQ)	max th.
WF+PF	2-(4-22)	47.4K
WF+PF opt	2-(4-25)	51.8K

Figure 8-27: *Left)*: Maximum throughput (results/s) obtained by the different patterns when using all available physical cores of the *KNL* platform. *Right)*: Number of workers used and maximum throughput of the different configurations with and without the *opt2* `WindFlow` optimization flag.

In the left-hand side of Figure 8-26 is shown the sustained throughput of the `Pane_Farm` pattern for 1 minute execution time in the stable state. For the configuration tested (R = 500Ktuples/s, w = 1000ms s = 2ms), both the PF and the "WF+PF" patterns are able to reach the maximum throughput (given by w/s). The fluctuations around the maximum value are given by small variations in the input data rate introduced by the generator stage. The test demonstrates that the "PF opt" and "WF+PF opt" do not introduce significant differences in the offered throughput even if there are no bottlenecks in the system. It is worth pointing out that the "opt" versions use a lower number of nodes. For example, the "WF+PF" uses 34 nodes while the "WF+PF opt" configuration uses 28 nodes (2 PF replicas, 2 PLQ Workers and 10 WLQ Workers).

Finally, Figure 8-27 shows the results obtained considering the count-based version of the Skyline test-case. In this test, the aim is to measure the maximum throughput sustained by the PF and the "WF+PF" patterns with and without the graph transformations. All configurations use the same number of nodes (64 nodes, including the generator and the collector nodes), this means that in the "opt" version it is possible to use more Worker nodes. The results obtained are interesting. Basically, they support two claims: 1) a proper pattern nesting can improve the sustained throughput of the application (the “WF+PF” pattern offers about 10% more throughput than the “PF opt”) ; 2) the building block transformations provide an extra performance improvement of about 7% because they allow reducing the number of service nodes thus making room for using more Worker nodes on the available resources.

8.4 Summary

In this chapter, we introduced the new FastFlow software component implementing concurrency graph transformation mechanisms. We studied different transformations aiming at reducing both the size of the graph and to remove sequential building block that potentially may introduce bottlenecks in the process network implementing the application. The new software layer provides a rich API implementing several transformations. It also offers simple interface functions that can be used by the application programmer to apply the most straightforward transformations automatically without compromising the application’s functional semantics.

Basically, the new FastFlow software layer described in this chapter offers two distinct interfaces targeting different programmers: 1) a set of “low-level” functions implementing several different graph transformations targeting RTS programmers; 2) a simple interface function (`optimize_static`) capable of applying a subset of all available transformations automatically and transparently to the user.

To evaluate the impact of the proposed transformations (whether implemented automatically or directly by the RTS programmer), we used two existing parallel frameworks (namely PiCo and Peafowl), two streaming benchmarks (from the P^{3}ARSEC benchmark suite) implemented on top of the FastFlow library, and a new parallel library (`WindFlow`) which makes broad use of several building block transformations presented in this chapter. The results obtained demonstrate that the transformations implemented and the mechanisms proposed are able to improve the application performance and to significantly increase the resource usage efficiency by reducing the number of nodes of the graph.

Chapter 9

Parallel Patterns

9.1 Introduction

In this chapter, we discuss parallel patterns implemented on top of the FastFlow building block software layer. Specifically, we concentrate on *non-streaming* parallel patterns. The motivation for this choice is twofold:

(i) The FastFlow library has been conceived for streaming computations. Therefore, it already provides suitable abstractions for streaming. Besides, some new parallel research frameworks such as SPar [177, 105] and PiCo [250, 251] have been developed for targeting both classical streaming computation and new Big Data Analytics applications, respectively. Both frameworks use the FastFlow library as RTS, specifically, they make extensive use of the *pipeline* and *farm* building blocks. The PiCo framework is currently moving to the new version of FastFlow to take advantage of the new *all-to-all* building block.

(ii) In Chapter 7 and Chapter 8 we have already shown the effectiveness of the FastFlow building blocks and their composability for implementing streaming use-cases (e.g., video streaming, network-based computations, and Data Stream Processing patterns). Moreover, several recent research works from people of the Parallel Computing Group of the University of Pisa targeted the implementation of complex parallel patterns for high-performance Data Stream Processing on multi-cores by using as RTS the FastFlow building blocks [126, 246, 247]. The WindFlow library introduced in Chapter 8 is the software result of these research efforts.

The stream-oriented high-level parallel patterns provided by the FastFlow library such as ff_Pipe, ff_Farm and ff_OFarm are all implemented directly on top of ff_pipeline and ff_farm building blocks. The main differences of these high-level patterns with respect to the corresponding building blocks are mainly related to the interface provided to the user, which is more user-friendly and easier to use. As an example, the ff_Farm pattern by default has the Collector entity, and it can also be instantiated by passing a C-like function for implementing the business logic of the Worker without the need of creating the vector of Worker nodes (see also Section 7.3.2).

In this chapter, we aim at demonstrating the high flexibility of the FastFlow building blocks that can also be profitably used for implementing ready-to-use and efficient task-based parallel computations that do not natively manage data streaming. We will discuss in details three different patterns: 1) the *ParallelFor* pattern providing a versatile implementation of the *Map* and *Map+Reduce* abstractions; 2) the *Macro Data-Flow* pattern modeling general, non-recursive, parallel computations by automatically managing data-flow dependencies among tasks; and 3) the *Divide & Conquer* parallel pattern modeling classical recursive computations.

The *ParallelFor* pattern has been initially implemented within the ParaPhrase project [187] and its interface was based on C-macros [119]. Here we present the new version using a modern C++ interface implemented within the RePara project (cf. Section 3.3.5). The *Macro Data-Flow* pattern was initially implemented to solve stencil-based numerical kernels [69] then it has been re-engineered to be used for general use-cases. Finally, the *Divide & Conquer* pattern has been implemented within the RePhrase project and added to the latest stable version of the FastFlow library [109].

To assess the usability of the three patterns presented in this chapter, we will use well-known algorithms, such as the Mergesort algorithm and the Cholesky factorization algorithm. Finally, to assess the performance of the FastFlow implementation, we compare with state-of-the-art frameworks offering either native high-level constructs (i.e. parallel-for) or general implementation mechanisms (i.e. the task-based programming model) for implementing the application kernels considered. Each test reported in the following sections has been run multiple times, and the average results are shown. The standard deviation measured was typically low and it is not reported in the plots for readability reasons.

9.2 ParallelFor and ParallelForReduce

Data parallelism is a parallelization paradigm where a large input collection of data elements is computed by processing independent sub-collections (or partitions) of data elements in parallel. The input collection (that can also belong to a bounded or unbounded stream) is split into multiple partitions each computed in parallel by applying the same function to each partition (possibly to each item of the partition). The results produced are collected in one single output collection, usually having the same type and size of the input (this is the case for the *Map* pattern). The primary objective of data-parallel computation is to reduce the *completion time* (latency) of the entire computation on the input collection.

The computation on the sub-collections may be completely independent, meaning that the computation uses only data coming from the current sub-collection, or, instead may be dependent on previously computed data. In the latter case, the function applied to the sub-collections might have an internal state. The way input data is partitioned and assigned to executors may introduce *workload imbalance* during the computation due to the potential variable calculation time associated to each distinct partition. Numerous techniques and algorithms have been proposed to define static and dynamic assignment of partitions to processing elements, among these the *work-stealing* algorithm is the most widely used [52].

Data parallelism can be specified at different levels of granularity and implemented at thread level (Thread Level Parallelism – TLP) or at the instruction level (Instruction Level Parallelism – ILP). To increase throughput, ILP can be used to process several elements simultaneously within each partition assigned to multiple threads.

One of the main sources of data parallelism are loops and in general iterative computations, where consecutive iterations working on independent or read-only data can be executed in parallel. A sequential iterative kernel having independent iterations is also known as a *parallel loop*. Parallel loops may be clearly parallelized by using the *farm* pattern/building block by *streaming loop's iteration* [18]. However, this might require a substantial refactoring of the original loop code running the risk to introduce bugs and also not preserving the *sequential equivalence*, a property often desired by many programmers. Also, the selection of the appropriate parallel pattern together with a correct implementation of the sequential wrapper code and the right choice of the scheduling policy are all critical aspects for obtaining the best performance.

To alleviate the effort required by the programmer, the FastFlow library provides the `ParallelFor`, `ParallelForReduce` and the `ParallelForPipeReduce` patterns implemented on top of FastFlow building blocks to simplify the parallelization of loops with independent iterations. The approach followed for implementing these patterns mimics the one proposed by other parallel programming frameworks such as OpenMP [102, 84], Intel TBB [282] and Cilk [51]. In the following, we will refer to `ParalleFor*` as a wider class that includes all three patterns and their implementations.

9.2.1 Pattern definition and implementation

The FastFlow `ParallelFor*` patterns can be used to parallelize loops having independent iterations.

The class interface provides a set of *parallel_for* methods, which differ in the number of arguments and for the signatures of the body function. The loop body may be a standard function or a C++ lambda-function. A single `ParallelFor*` object instance can be used as many times as needed, for example inside a sequential loop or in multiple invocations with different loop body provided via C++ lambdas. Nested invocations of the *ParallelFor** methods are not supported.

Let us consider the `ParallelFor` class. Its constructor accepts three (optional) arguments:

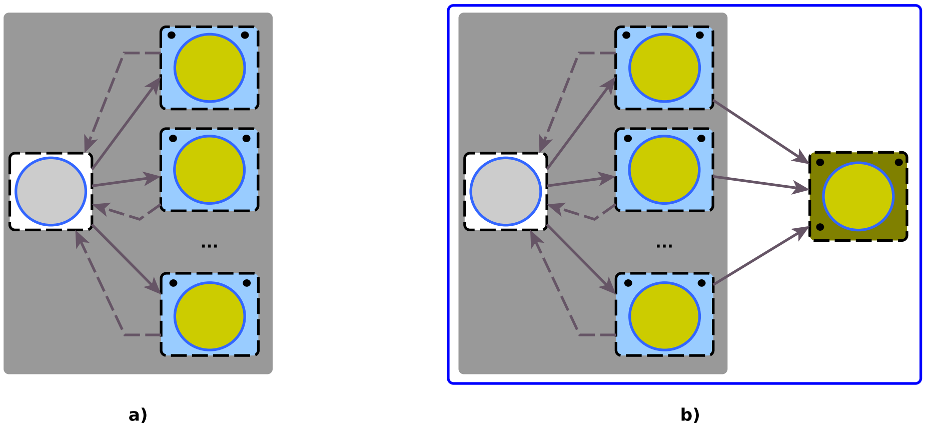
```
ParallelFor(const long maxnworkers=FF_AUTO,
            bool spinWait=false,
            bool spinBarrier=false);
```

The first argument is the maximum number of Workers that can be used in one single invocation of the `parallel_for` method, i.e. this is the maximum *parallelism degree* of the pattern. The default value uses all available cores of the platform where it is executed. The second argument configures the *non-blocking* concurrency mode between two different invocations of the object methods. This means that the RTS threads are not suspended at the end of the computation. The third parameter configures the *non-blocking* barrier implementation. These two last flags are important mainly for performance reasons when the pattern is used inside an iterative loop, i.e. when the *ParalleFor* is called multiple times.

In the following snippet of code, a few usage examples of the *parallel_for* methods are shown. The `bofyF*` functions provide different signatures of the loop body; `first` and `last` define the iteration range; `step` the step size of the iterating index; `grain` and `nworkers` define the number of iterations assigned to a Worker and the number of Workers used, respectively.

```
ParallelFor pf; // object instance
auto bodyF1=F(const long i);
auto bodyF2=F(const long i, const int worker-id);
auto bodyF3=F(const long begin,const long end,const int worker-id);
pf.parallel_for(first,last,bodyF1);
pf.parallel_for(first,last,bodyF1,nworkers);
pf.parallel_for(first,last,step,bodyF1,nworkers);
pf.parallel_for(first,last,step,grain,bodyF1,nworkers);
pf.parallel_for_thid(first,last,step,grain,bodyF2, nworkers);
// explicit management of internal loop on sub-partitions
pf.parallel_for_idx(first,last,step,grain,bodyF3,nworkers);
```

The methods `parallel_for_thid` and `parallel_for_idx` are just two "lower-level" interfaces of the `parallel_for` method, that allow us to access, from within the loop body, the `worker-id` of the Worker executing the body function, and to directly execute (by means of a user-defined local loop) the loop iteration range assigned to the Worker, respectively. In the `parallel_for_idx` version, the local iteration range is given by the interval `(begin, end)` determined by the arguments of the body function `bodyF3`. These lower-level versions greatly increase the flexibility of the pattern and they can also be used for debugging purposes.



a) b)

Figure 9-1: Building blocks implementation of the *ParallelFor*, *ParallelForReduce* (a) and of the *ParallelForPipeReduce* (b).

The *ParallelFor* and *ParallelForReduce* patterns are implemented using a *farm* building block while the *ParallelForPipeReduce* is implemented using a two-stage *pipeline* whose first stage is a *farm* and the second stage is a sequential *multi-input* node (see Figure 9-1, left-hand side and right-hand side, respectively). The *ParallelForPipeReduce* pattern has been effectively used in the parallelization of the PWHATSHAP framework [60].

Dot product example. The *dot product* performs a pairwise multiplication of two vectors of the same length and then sums up the intermediate result, i.e. given A and B of length N: $d = \sum_i^N A[i] \cdot B[i]$. This simple kernel, part of the *BLAS* library, can be implemented by using a `ParallelForReduce` pattern as sketched in Code 24.

The *ParallelForReduce* object producing a `double`-precision result (stored in the `sum` variable), is create at line 5 with `nworkers` Workers. The `SpinWait` flag is

```
1  #include <ff/ff.hpp>
2  #include <ff/parallel_for.hpp>
3  using namespace ff;
4  initVectors(A,B);
5  ParallelForReduce<double> pfr(nworkers,(nworkers<ff_numCores()));
6  auto Fsum = [](double& v,const double& elem) {v += elem;};
7  double sum{0.0};
8  pfr.parallel_reduce(sum, 0.0,
9        0, vectorLength,1,chunksize,
10       [&](const long i,double& sum) {sum += A[i]*B[i];},
11       Fsum);
```

Code 24: FastFlow program computing the dot product of two vectors using the `ParallelForReduce` pattern.

enabled only if the number of Workers is greater than the number of machine cores. The loop body (line 10) is executed in parallel by all Workers. Each one updates a local per-Worker reduction variable executing the product operation between elements of index `i`. The final reduce function, defined at line 6 and executed sequentially, updates the reduction variable `sum` with the partial results computed by all Workers. The *chunksize* (line 9 defines the number of consecutive iterations assigned to each Worker (i.e. the computation granularity).

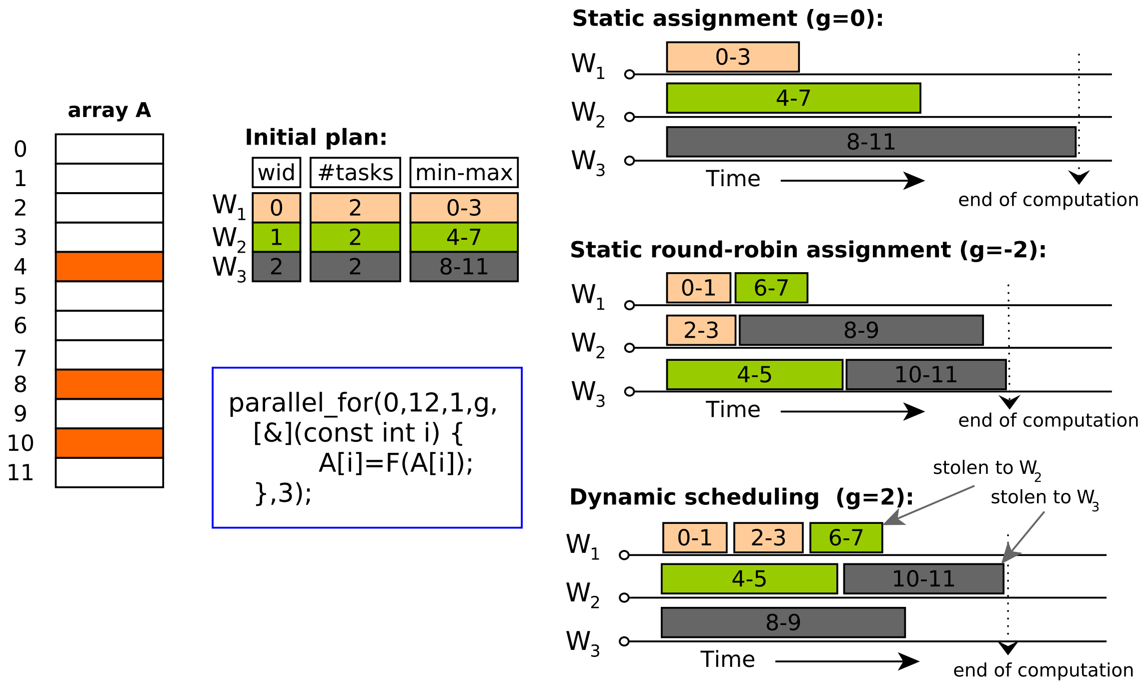
Iteration scheduling. The loop iterations are assigned to the Workers according to a static or dynamic scheduling policy. Three distinct iteration scheduling policies are currently implemented in the `ParallelFor*` patterns, two static policies and one dynamic.

1. **static scheduling**: the iteration space is (almost) evenly partitioned in large contiguous chunks, and then they are statically assigned to Workers, one chunk for each worker.

2. **round-robin scheduling with interleaving** k: the iteration space is statically divided among all active Workers in a round-robin way using a stride of k. For example, to execute 10 iterations (from 0 to 9) using 2 Workers and a stride $k = 3$, then the first Worker executes iterations $0, 1, 2, 6, 7, 8$ while the second

Worker executes iterations $3, 4, 5, 9$. The default static scheduling is obtained by setting a stride $k = iterationspace/nworkers$.

3. **dynamic scheduling with chunk size** k: in this case no more than k contiguous iterations at a time are dynamically assigned to be computed to one of the active Workers. As soon as a Worker completes the computation of one chunk of iterations, a new chunk (if available) is selected and assigned to the Worker. The FastFlow RTS tries to select as many contiguous chunks as possible in order to better exploit spatial locality. This allows having a good trade-off between iterations affinity and workload balancing.



Initial plan:

	wid	#tasks	min-max
W_1	0	2	0-3
W_2	1	2	4-7
W_3	2	2	8-11

```
parallel_for(0,12,1,g,
  [&](const int i) {
        A[i]=F(A[i]);
  },3);
```

Figure 9-2: Example of the different iteration scheduling strategies provided by the *ParallelFor** patterns.

The three iteration scheduling policies are exemplified in Figure 9-2. By default, the *static scheduling* is used for all *ParallelFor** patterns. In general, the scheduling policy is selected by specifying the `grain` parameter of the `parallel_for` method. If the `grain` parameter is not specified or if its value is zero, then the *default static scheduling* is selected. If `grain` is greater than zero, then the *dynamic scheduling* is selected with $k = grain$. Finally, to use the *round-robin scheduling with interleaving*

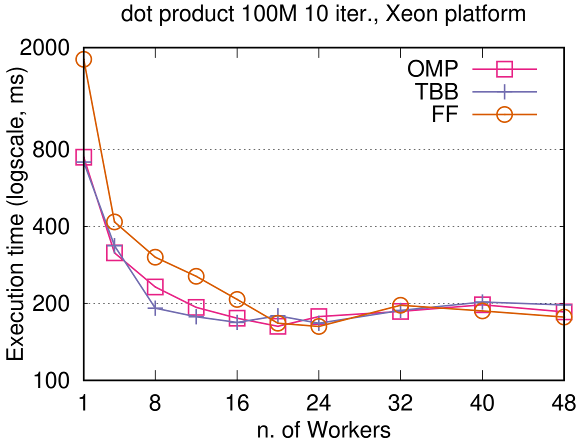
k a value less than zero has to be set for the `grain` parameter.

9.2.2 Evaluation

In this section, we evaluate the performance of the `ParallelFor*` implementations, we consider both simple benchmarks (the *dot product* and an *unbalanced* parallel loop computation) as well as complex applications coming from the PARSEC 2.0 benchmark suite [47].

Simple benchmarks. The first benchmark we consider, is the well-known *dot product* operation. We tested the kernel on both the *Xeon* and *KNL* platforms, comparing the FastFlow `ParallelForReduce` implementation with both an OpenMP *#pragma*-based *parallel-for* with static scheduling and Intel TBB *parallel_reduce* implementations of the same kernel. The OpenMP version is the one offered by the `gcc` compiler version 6.4.0 and 7.3.0 for the two platforms, respectively. The Intel TBB versions used are the version 4.1 for the *Xeon* platform and the one shipped with Intel Parallel Studio XE 2017 for the *KNL* platform.

The results obtained for vectors of size 100 million double precision elements, are shown in Figure 9-3 (the time refers to 10 consecutive iterations of the *dot product*). To have a fair comparison among different versions, all of them use the same number of RTS threads. This means that both the OpenMP version as well as the TBB version use a number of threads that is equal to the number of Workers plus one (i.e. $n.ofWorkers+1$ in the figures). On the *Xeon* platform, the FastFlow implementation is initially slower than the other two implementations because of the different number of real executors. Starting from 20 Workers, where the execution time is about the same for the three versions (163ms, 178ms and 167ms for OpenMP, TBB and FastFlow, respectively), the FastFlow implementation obtains slightly better results (see also the detailed results reported in the right-hand side of Figure 9-3). In Figure 9-4 are shown the results obtained running the *dot product* benchmark on the *KNL* platform. Also in this platform, the three versions obtain very similar results with a maximum speedup value of 86, 85 and 81 for OpenMP, TBB and FastFlow, respectively (the



#Workers	OMP	TBB	FF
21	165	169	160
22	173	175	156
23	186	174	154
24	177	167	163
25	186	183	189
26	188	181	187
27	188	186	188

Figure 9-3: Left:) Execution time (log scale, milliseconds) of the dot product computation repeated 10 times of two vectors of 100 million double precision elements executed on the *Xeon* platform. Right:) Detailed execution times (milliseconds) close to the minimum value.

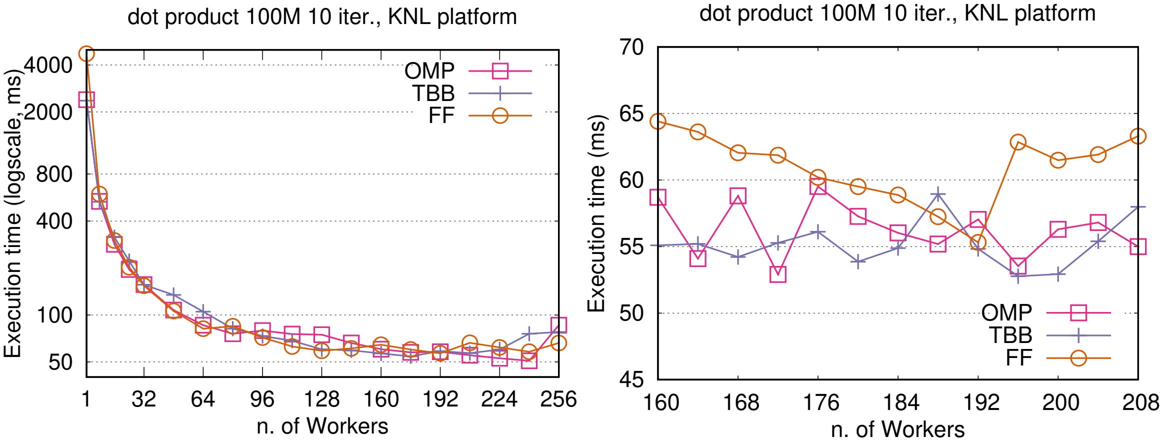


Figure 9-4: Left:) Execution time (log scale, milliseconds) of the *dot product* computation repeated for 10 iterations of two vectors of 100 million double precision elements executed on the *KNL* platform. Right:) Zoom of the execution times (milliseconds) around the minimum value.

sequential execution time on the *KNL* platform is 4.48s). On the right-hand side of Figure 9-4 it is shown the execution time in the range 160 − 208 Workers where all versions obtain the minimum value.

In the left-hand side of Figure 9-5 is shown the comparison of the execution time obtained by the *ParalleForReduce* (PF) and *ParallelForPipeReduce* (PFPR) of the dot-product benchmark on the *Xeon* platform. For this simple application, there is no benefit in performing the reduce part in pipeline with the map part. Therefore

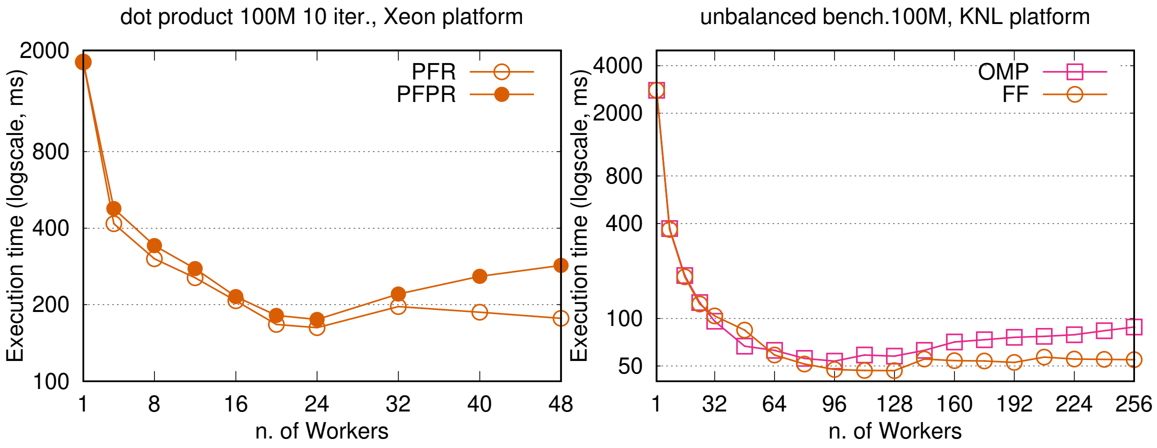


Figure 9-5: Left:) Comparison between the *ParallelForReduce* (PFR) and the *ParallelForPipeReduce* (PFPR) using the dot product benchmark on the *Xeon* platform. Right:) Execution time of the unbalanced loop benchmark on the *KNL* platform. Comparison between the *ParallelFor* pattern version with and without the active scheduler and the OpenMP version using the dynamic scheduling.

the *ParallelForPipeReduce* version obtains slightly worse results. However, up to 24 Workers, the two implementations obtain similar results (the difference is less than 2%), then the extra threads used to execute the reduce part introduce only overhead with no benefit.

To evaluate the overhead of the *dynamic scheduling* policy implemented by the *ParallelFor** patterns, we used a simple synthetic benchmark. A function F is applied to all elements of a vector A. The vector has N elements in which only a small subset of entries have a high computational cost while the remaining elements have zero computation time. The most expensive entries (having different cost granularities) are all placed at the beginning of the vector. In this simple test, assigning more than one iteration to one Worker may result in a significant load unbalancing. The best strategy (among those available) is to use the dynamic scheduling with a computation *grain* of 1 iteration. We compare the FastFlow pattern against the OpenMP version using the *dynamic* scheduling by setting the environment variable `OMP_SCHEDULE="dynamic,1"`. The results obtained on the *KNL* platform are reported in the right-hand side of Figure 9-5. As can be seen, the dynamic scheduling of the *ParallelFor** pattern obtains almost the same results as the one implemented in the OpenMP RTS up to 80 Workers, and then it exhibits a more stable trend with increasing number of

Workers.

PARSEC applications. PARSEC [47] (Princeton Application Repository for Shared-Memory Computers)[1] is a collection of various multi-threaded programs with high system requirements that have been widely used to test the performance of multi/many-cores. It covers a wide set of application domains comprising 13 programs from different areas of computing. Each application is provided with several input sets. The *native* dataset is representative of a realistic execution scenario of each application. Here we consider a sub-set of PARSEC applications belonging to the data parallelism domain and on large data structures logically partitioned among multiple threads (`blackscholes`, `bodytrack`, `facesim`, `fluidanimate`, `raytrace`, `streamcluster`). All PARSEC benchmarks have been implemented using FastFlow parallel patterns in De Sensi et al. [129]. The six benchmarks considered here can all be parallelized using one or more *ParallelFor** patterns. For all kernels tested is available a native PThreads implementation, when available we compare to the OpenMP and Intel TBB versions shipped with the benchmark suite.

blackscholes. The application belongs to the Intel RMS benchmark suite (Recognition, Mining and Synthesis). It performs pricing for a portfolio of European options by numerically solving the Black-Scholes partial differential equations. The PThreads implementation divides the portfolio into work units, one for each available thread. Then, each thread calculates the prices for the options in its work unit. This algorithm is iterated multiple times to obtain the final estimation of the portfolio. This benchmark is an iterative data-parallel computation that can be implemented by using a *ParallelFor* within a sequential loop.

bodytrack. This application is aimed at tracking the body pose of a human subject by analyzing videos collected by multiple cameras. A *frame* contains one *image* from each camera. `bodytrack` has basically two phases that are executed for each frame. In the first phase, three kernels are executed for each image. After this phase, two additional kernels are applied a number of times on the frame. Before applying a

[1] We refer to the PARSEC version 3.0: `http://parsec.cs.princeton.edu/overview.htm`

kernel, we need to ensure that the previous kernel is terminated. Accordingly, we can exploit parallelism only within each kernel. The PThreads version is implemented by using a thread pool, which can execute different kernels. The execution starts in the main thread and, for each frame, when a kernel needs to be executed the main thread sends a command to the pool with an identifier corresponding to the kernel type. The threads in the pool will then start to process chunks of the frame with the specified kernel. To keep the load balanced, the chunks are not statically partitioned. Each thread, after the processing of the current chunk, accesses a shared variable (using a lock) to get the identifier of the next chunk, and updates such variable. In the pattern-based implementation, the application can be parallelized by using multiple *ParallelFor*.

facesim. It is an Intel RMS application simulating the motion of human faces. It applies the iterative Newton-Raphson algorithm over a sparse matrix. At every time step, different kernels are executed on a mesh (some kernels are executed multiple times within a single time step). The PThreads version uses a thread pool which, at every time step, executes different kernels on the mesh. Every time a kernel is found during the execution, it is executed by the thread pool, where each thread works on a statically assigned portion of the mesh. As for the `bodytrack` application, `facesim` can be parallelized using multiple *ParallelFor* sequences repeated several times.

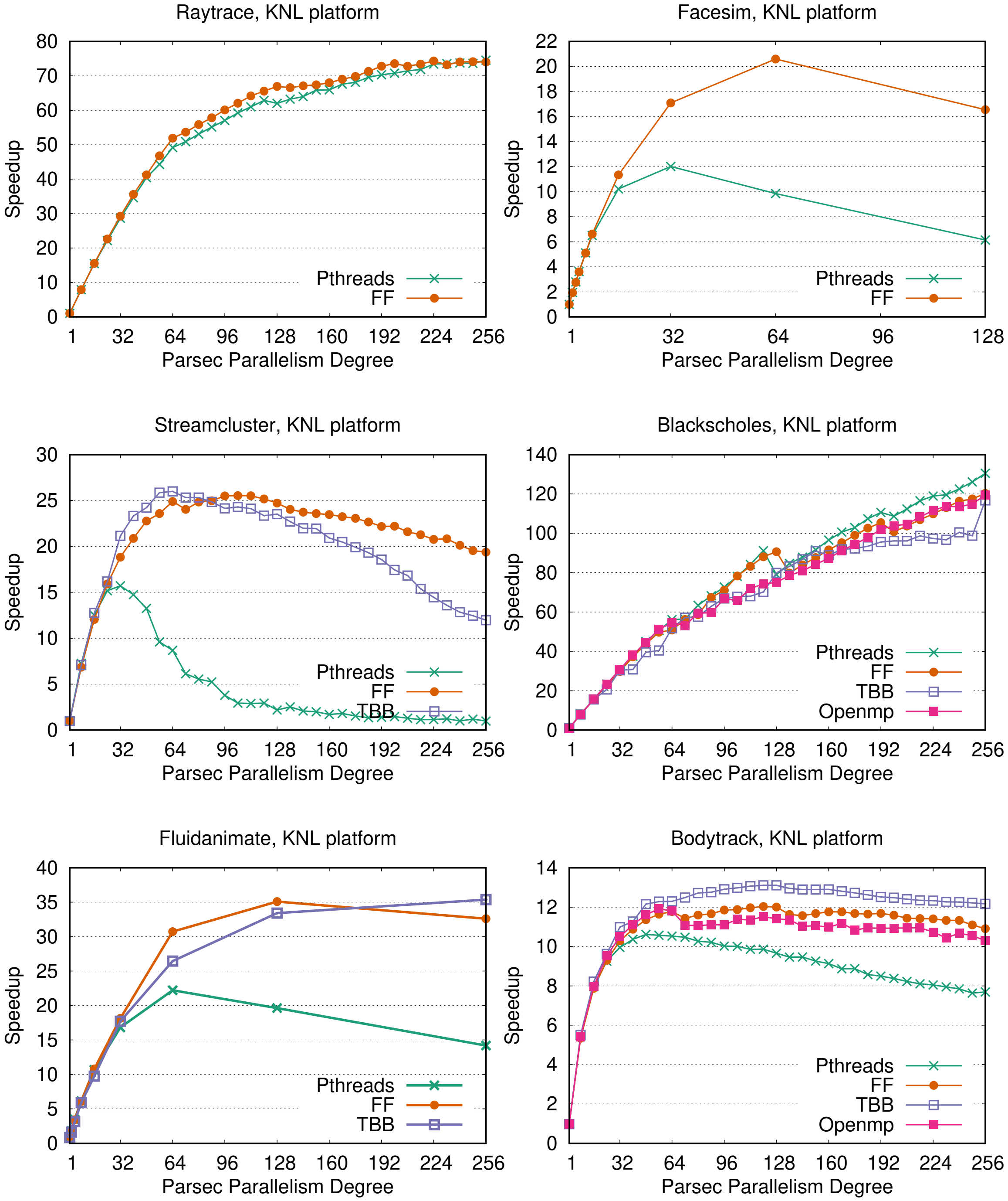
fluidanimate. It is another Intel RMS benchmark that uses an extension of the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid. At every time step, the application executes nine kernels to compute the position of the fluid particles at the next time step. As in other benchmarks, the sequence of kernels is sequential while parallelism can be safely exploited within each kernel region. In the PThreads implementation the three-dimensional space is statically divided among the threads. Each thread applies each kernel on its space partition. A barrier is executed by all the threads between two successive kernels. In the pattern-based implementation each kernel can be implemented as a *ParallelFor* pattern.

raytrace. This application consists in a graphical render aimed at generating animated 3D scenes by using a hierarchical grid raytracing algorithm. A kernel is executed at each frame. In the PThreads version the kernel is parallelized by partitioning the 3D scene among the threads. The work is dynamically partitioned and, similarly to the `bodytrack` PThreads implementation, once a thread finishes to process a partition, it gets another one in order to keep the load balanced. The application can be modeled as a single *ParallelFor* iterated a fixed number of times. Differently from `blackscholes`, the computation is extremely unbalanced and a good dynamic scheduling of the partitions is of great importance.

streamcluster. It is an application that solves the online clustering problem over incoming streaming data. The program consists in a sequence of loops whose iterations can be executed in parallel. Different loops are executed sequentially by using barriers, and they are interleaved by serial regions of code whose length impacts the overall speedup. The computational kernel consists of two phases. The first iterates a composition of a *Map+Reduce* and a number of *Map* instances (that are in turn iterated multiples times). The second phase, working on different data, repeats the same steps exactly one time. In the FastFlow implementation, the *Map+Reduce* instances are implemented with a *ParallelForReduce* pattern while the *Map* instance with a *ParallelFor*.

Figure 9-6 shows the speedup on the *KNL* platform of the six PARSEC applications considered. The time measured is the one spent in the so-called *region of interest* (ROI), which includes all parts sensitive to the parallelization. This approach is commonly adopted when comparing different parallelizations of the same application. Each program has been run multiple times, and the average results are the ones used to compute the speedup. All the benchmarks have been executed with the original parameters provided by PARSEC.

Except for the `blackscholes` applications, for all other ones the PThreads versions do not obtain the best speedup. In all cases tested, the FastFlow *ParallelFor** patterns provide a very good level of speedup that is comparable to the one obtained by the OpenMP and TBB implementations which provide quite optimized implemen-



Raytrace, KNL platform

Facesim, KNL platform

Streamcluster, KNL platform

Blackscholes, KNL platform

Fluidanimate, KNL platform

Bodytrack, KNL platform

Figure 9-6: Speedup of some data-parallel benchmarks of the PARSEC suite.

tation of the *parallel-for* pattern.

For `fluidanimate`, `streamcluster` and `facesim`, the Pthreads version performs poorly. For `fluidanimate`, this is mainly due to the different implementation of the barrier provided by the various frameworks. A barrier is executed after each parallel kernel. This performance difference is remarkable only at high concurrency levels. Regarding `streamcluster`, the parallelization based on the *ParallelFor* pattern allows us to simplify the code and remove synchronizations leading to a performance improvement up to 40% on the *KNL* platform. Finally, in the PThreads version of `facesim`, when a parallel kernel is found during the execution, one abstraction of a mesh partition is inserted for each thread in a shared queue, accessed by all threads and protected by `locks`. Instead, in the FastFlow implementation, a partition is statically assigned to each thread without any need to access any shared data structure, thus achieving better speedup.

Platforms	**Version**	BS	BD	FC	FL	RT	SC
Xeon	PThreads	29.4	7.9	7.3	10.6	23.9	9.8
	FastFlow	28.8	8.3	10.9	10.9	23.9	12.6
	TBB	26.5	9.2	-	10.9	-	13.4
	OpenMP	29.4	7.8	-	-	-	-
KNL	PThreads	131.3	10.6	12.1	22.2	74.4	15.7
	FastFlow	120.3	12.1	20.6	35.1	74.4	25.5
	TBB	117.4	13.2	-	35.4	-	26.0
	OpenMP	120.2	11.9	-	-	-	-
Power	PThreads	31.5	8.5	6.5	9.8	-	5.4
	FastFlow	39.8	10.3	10.9	15.1	-	8.2
	TBB	35.5	9.0	-	11.0	-	7.5
	OpenMP	34.1	9.0	-	-	-	-

Table 9.1: Best speedup figures on all platforms considered. `BS` (`blackscholes`), `BD` (`bodytrack`), `FC` (`facesim`), `FL` (`fluidanimate`), `RT` (`raytrace`), `SC` (`streamcluster`). The `raytrace` benchmark can not be compiled on the IBM Power platform due to some architecture specific assembler instructions.

In Table 9.1 are reported the best speedup figures on the three platforms described in Table 6.1. Small differences and discrepancies in the results between different versions of the same benchmark and between different architectures are due to differences in the compiler, architecture and by the intrinsic differences and optimizations in the RTS of the frameworks used

To conclude, the *ParallelFor** patterns provided by the FastFlow library exhibits a comparable level of performance compared to state-of-the-art implementations, namely OpenMP and Intel TBB. This demonstrates the good level of performance and flexibility of the FastFlow building blocks again. Moreover, though the usability of the OpenMP pragma-based approach is undoubtedly superior, the FastFlow library provides a comparable level of usability of other high-level library-based approaches with the advantage of simpler code refactoring and thus wider design space possibilities [129].

9.3 Macro Data-Flow

The *Data-Flow* programming paradigm models a parallel program as a directed acyclic graph (DAG) where nodes are operations while edges model data dependencies. Operations are functional units of computation that are executed as soon as all input data elements are present.

Parallelism can be expressed at the finest granularity level (i.e. simple instructions). This is at the expense of considerable complexity in the development of the implementation model, specifically, in the management of data dependencies and storage space (where operands and meta-data are maintained) and in the efficient detection and scheduling of the firable (ready to be executed) instructions. As a consequence, real hardware implementations of this paradigm usually provide lower scalability and performance if compared to the control-driven execution model [319].

Instead of expressing parallelism at the instruction level, portions of the sequential code having pure functional dependencies between input parameters and output results, are considered *Macro Data-Flow* (MDF) instructions. The resulting MDF program is therefore represented as a graph whose nodes are coarse-grained computational kernels and edges *read-after-write* data dependencies. The execution of a MDF program starts with the assignment of the input data ("tokens" in data flow jargon) to the first node in the MDF graph. The RTS is in charge of executing the MDF graph by scheduling fireable macro-instructions (i.e. tasks) to a set of anonymous executors (usually called Workers) and managing data dependencies.

Stream parallelism and data parallelism are handled in different ways. Stream parallelism is managed by creating, for each item appearing on the input stream, a new "fresh" copy of the MDF graph and passing the input element as the input token of the graph. Therefore stream parallelism derives from the execution of fireable instructions from different graph instances. This requires labeling of graph instructions with additional tags [116]. Data parallelism is managed by inserting into the graph instructions which decompose their input data structure(s) into multiple disjoint data sets and direct these data sets to a number of independent MDF instructions computing partial results which are eventually directed to another instruction which has as input the partial results and then computes the final result [69].

From the RTS standpoint, the following three points represent important aspects for an efficient implementation of a MDF interpreter:

Construction of the task graph: In the general case the task graph could be very large and its generation time can affect the overall computation time. In addition, the memory required to store the entire graph may be very large. To alleviate these issues, a widely used solution consists in generating the graph during the computation, such that only a "window" of the graph is maintained in memory. This also allows overlapping tasks computation with the graph generation.

Handling task dependencies: It is possible to identify two main operations on the task graph: 1) update data dependencies after the completion of previously scheduled tasks; 2) determine ready (fireable) tasks to be executed. These operations have to be executed in parallel with tasks computation.

Scheduling of ready tasks: A task having all input dependencies resolved may be selected for execution. This selection needs to be performed in a smart way considering that two main optimizations can be applied in this phase when targeting multi-core architectures: i) *locality optimization* in order to better exploit cache level hierarchies, and ii) *parallelism optimization* in order to maintain the number of ready tasks as big as possible during the execution to prevent stalls. The first optimization is based on the observation that graph edges represent data dependencies, so that if task i unlocks execution of task j, then they share at least one of the dependencies.

Executing task j on the same processor that has executed task i, increases the probability that the common data reside in the cache hierarchy. A common approach to exploit this data locality is to maintain ready queues per processor, and implement a *work-stealing* scheduler. The second optimization relies on the fact that a graph node with a higher degree of output edges, might unlock a larger number of tasks. Following this principle, a possible heuristic is to select with higher priority among all fireable tasks those that have the highest number of outgoing edges. This can be accomplished by ordering fireable tasks with respect to the degree of the related node on the graph.

Finally, to reduce memory consumption, in-place computation is generally used on shared memory platforms. This implies that, the graph has to be enriched by additional anti-dependencies (*write-after-read*) between tasks for removing the need of costly copies of the original data structure. This at the price of a possible lower parallelism between MDF instructions.

The FastFlow MDF pattern [20, 69], implemented by the C++ class `ff_mdf`, aims to provide the programmer with a high-level implementation of the Macro Data-Flow model working both on a stream of elements and on a single data-parallel computation. The pattern hides all complex implementation details related to the scheduling of fireable tasks and the efficient management of data dependencies. It can also be used as one of the stages of a *pipeline* pattern (`ff_Pipe`) or as a Worker of a *farm* pattern (`ff_Farm` and `ff_OFarm`).

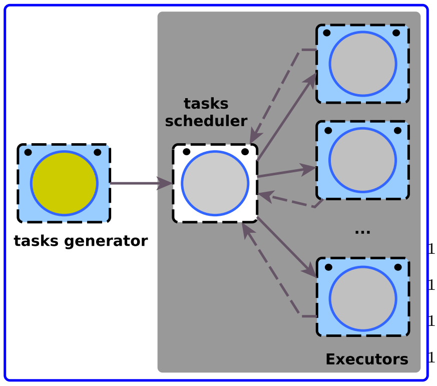
9.3.1 Pattern definition and implementation

A MDF interpreter is logically composed of three distinct entities: i) one entity generating the tasks by executing the user algorithm that eventually produces the instructions for building the task graph; ii) a scheduler that manages the data dependencies among tasks and schedules ready ones to a pool of executors; iii) a pool of anonymous Workers (i.e. the real MDF interpreters) executing the tasks and notifying their completion (i.e. partial results) to the scheduler. The generated tasks and the partial results of the computation can be modeled as a stream of data elements containing references to data stored in the shared memory.

The scheduler receives, in a non-deterministic way, both new tasks coming from the task generator and also completed tasks coming from the set of Workers. The task-graph of MDF instructions can be generated either statically or dynamically. For a very large graph, the entire static generation of all the graph is not feasible. From the sequential order in which tasks are generated by executing the user's algorithm, the scheduler computes a *partial ordering* that ensures computation correctness, and, by evaluating data dependencies among tasks, it adds the corresponding node/edge to the DAG structure. A completed task coming from one of the workers may either activate new tasks ready to be scheduled for execution or trigger the termination condition.

Building blocks implementation. The FastFlow implementation of MDF pattern uses a two-stage pipeline composition: the first stage is a sequential building block that executes the user's algorithm and produces a stream of tasks in output; the second stage is a *farm* building block where the Emitter executes the code of the scheduler and the Workers are the anonymous MDF interpreters. The Emitter node performs the pre-processing of the input tasks received by the first stage of the pipeline, and it builds the task-graph structure. The Workers are standard sequential building blocks which compute the results and route them back to the Emitter node through a feedback channel. On the left-hand side of Figure 9-7 is shown the logical schema of the building block implementation of the MDF pattern.

The user of the FastFlow MDF pattern has to provide a kernel code which generates macro-instructions. Input and output data of each macro-instruction must be clearly identified. As an example, in the right-hand side of Figure 9-7 is shown the simplified code of a simple example that uses the `ff_mdf` parallel pattern for computing in parallel $C[i] = A[i] + B[i]$ where $i \in [0..N[$. In this example, there are N independent tasks. The programmer provides a function (the function `Algo` in Figure 9-7) which describes the sequential algorithm that will be executed by the tasks generator stage (i.e. the first stage of the pipeline). The `AddTask` procedure (implemented by the



```
1  void Fun(int a, int b, int* c) {*c=a+b;}
2  void Algo(Parameters& P) {
3   auto mdf = P.mdf;
4   for(int k=0;k<P.N;k++) {
5     param_info _1={ID1+k, INPUT};
6     param_info _2={ID2+k, INPUT};
7     param_info _3={ID3+k, OUTPUT};
8     std::vector<param_info> pr{_1,_2,_3};
9     mdf->AddTask(pr,Fun,P.A[k],P.B[k],&P.C[k]);
10  }
11 }
12 int main() {
13   Parameters P{ ...};
14   ff_mdf mdf(Algo,P,nworkers);
15   if (mdf.run_and_wait_end()<0) return -1;
16 }
```

Figure 9-7: Left:) Building blocks used to implement the MDF pattern. Right:) A simplified example on how to instantiate and execute the FastFlow MDF pattern executing the tasks produced by the user's function `Algo` implementing a simple parallel loop.

MDF pattern) is used to produce a new task each time it is invoked. In the example, the macro-instruction is given by the function `Fun` (line 1). For each parameter of the task function, the user must specify a unique *identifier* and its *mode*. The *identifier* can be, for example, the memory pointer of the parameter. The *mode* specifies the directionality of the parameter: INPUT or OUTPUT. A special *mode*, called VALUE, is required for those parameters that are directly evaluated inside the task generator and do not concur to the DAG construction. All tasks generated via the `AddTask` function are packed and sent to the Emitter node, which creates the corresponding task-graph node.

In the MDF implementation, only the Emitter node of the *farm* building block works on the DAG, so that neither critical sections nor cache invalidations overhead is spent on updating the graph structure. The Worker threads receive tasks from the Emitter and then they send back the notification that the task has completed. The Emitter node non-deterministically receives completed task notifications from the set of Workers *and* new macro instructions, produced by the *AddTask* function, from the

```
task_t* svc(task_t* task) {
 if (fromInput(task)) {
   insertTask(task);
   schedule_tasks();
   if (graphSize()>Threshold) {
     // stop receiving from previous stage
     input_active(false);
   }
   return GO_ON;
 } // from previous stage
 handleCompletedTask(task);
 schedule_task();
 // restart receiving from previous stage
 if (graphSize()<Threshold) input_active(true);
 if (noMoreTasks()) return EOS;
 return GO_ON;
}
```

Code 25: Pseudo-code executed by the MDF task scheduler.

first stage of the pipeline. Since the memory required to store the entire DAG may be huge, the Emitter maintains in the main memory only the active portion of the DAG data structure implemented with an hash-table. When the number of generated graph nodes reaches a predetermined threshold (that can be tuned by the user), the input channel coming from the first stage of the pipeline is temporarily disabled; this way the graph generation is halted and the Emitter handles only completed tasks coming from the Workers. When the number of available instructions falls below the threshold, the *farm* input channel is enabled. The pseudo-code executed by the Emitter node for handling tasks is sketched in Code 25.

Worker nodes receive tasks through a *request-reply* custom scheduling policy similar to the default *on-demand* protocol provided by the *farm* building block. This policy ensures good workload balancing without using more complex work-stealing techniques. When multiple fireable tasks are available, they are enqueued in a per-Worker local queue managed by the Emitter. The queue is implemented as a priority queue where the task priority is computed on the basis of the number of its forward dependencies in the DAG. When a new task has to be scheduled to one of the Workers, the one with higher priority is extracted from the queue and sent to the Worker.

By using the proposed implementation schema, the RTS overhead is bound mainly into the two sequential stages: the task generator node and the *farm* Emitter. While on the one hand this approach may, in principle, limit the scalability of the pattern with a large number of graph nodes, on the other hand, it allows us to simplify the design and debugging of the implementation and to produce a solution with reduced programming effort.

```
void Chol(fComplex *A,int nb,int bs){
 for(int k=0;k<=(nb-1);k++) {
   for(int i=0;i<=(k-1); i++)
     CHERK(&A[k*bs*bs*nb+k*bs],
           &A[k*bs*bs*nb+i*bs],
           &A[k*bs*bs*nb+k*bs]);
   CPOTF2(&A[k*bs*bs*nb+k*bs],
          &A[k*bs*bs*nb+k*bs]);
   for(int j=(k+1);j<=(nb-1);j++){
     for(int i=0;i<=(k-1);i++)
       CGEMM(&A[k*bs*bs*nb+i*bs],
             &A[j*bs*bs*nb+i*bs],
             &A[j*bs*bs*nb+k*bs],
             &A[j*bs*bs*nb+k*bs]);
       CTRSM(&A[k*bs*bs*nb+k*bs],
             &A[j*bs*bs*nb+k*bs],
             &A[j*bs*bs*nb+k*bs]);
   }
 }
}
```

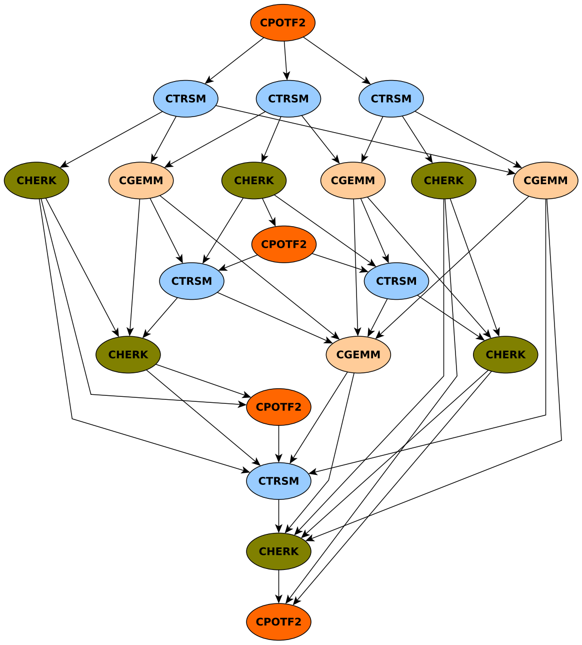


Figure 9-8: Left:) Cholesky factorization algorithm (left-looking variant, complex single-precision data). Right:) Resulting DAG of the factorization of a 4 × 4 tile matrix.

Cholesky factorization examples. In the right-hand side of Figure 9-8 is sketched the sequential Cholesky factorisation algorithm using the left-looking variant and operating on single precision complex data elements (i.e. float-based complex called `fComplex`). The function names are the ones used in the BLAS library. On the left-hand side of the same figure is shown the Cholesky graph instructions for a 4×4 time matrix.

The algorithm in Figure 9-8 is executed by the task generator stage. Instead of direct BLAS function calls, a macro instruction is generated by using the `AddTask` function in the same way reported in the example in Figure 9-7. For example, the `CPOTF2` call at line 7 produces a macro instruction generated as follows:

```
param_info _1={(uintptr_t)(&A[k*bs*bs*nb+k*bs]),INPUT};
param_info _2={(uintptr_t)(&A[k*bs*bs*nb+k*bs]),OUTPUT};
Param.push_back(_1); Param.push_back(_2);
mdf->AddTask(Param,CPOTF2,&A[k*bs*bs*nb+k*bs],&A[k*bs*bs*nb+k*bs]);
```

9.3.2 Evaluation

In this section, we evaluate the performance of the *Macro Data-Flow* parallel pattern. The first test we consider the implementation of a parallel loop to compare with the *ParallelFor* pattern. As test case, we used the loop showed in the right-hand side of Figure 9-7 where the function *Fun* executes a synthetic computation of a few thousands of clock cycles. The number of iterations is $N = 10,000$. The second kernel tested is the matrix multiplication ($C = A \times B$) by using Strassen's algorithm. The algorithm splits the A, B and C matrices in four disjoint parts (sub-matrices) and executes intermediate operations on the four parts to compute the final C matrix. The operations and the dependencies graph of the Strassen's algorithm is shown in Figure 9-9.

The results of the first two tests, parallel loop and Strassen, on the *Xeon* multi-core are reported in Table 9.2 left-hand side and right-hand side, respectively. For the parallel loop test, the *MDF* implementation obtains performance figures close to the ones obtained by the "specialized" *ParallelFor* pattern. For the Strassen algorithm, the maximum speedup that is expected to obtain simply executing the macro-instructions in parallel is about seven (there are seven matrix multiplication macro-instructions, see the DAG in Figure 9-9). The *MDF* version obtains a speedup of about six using seven Workers. Then, we tested the case in which the single macro-

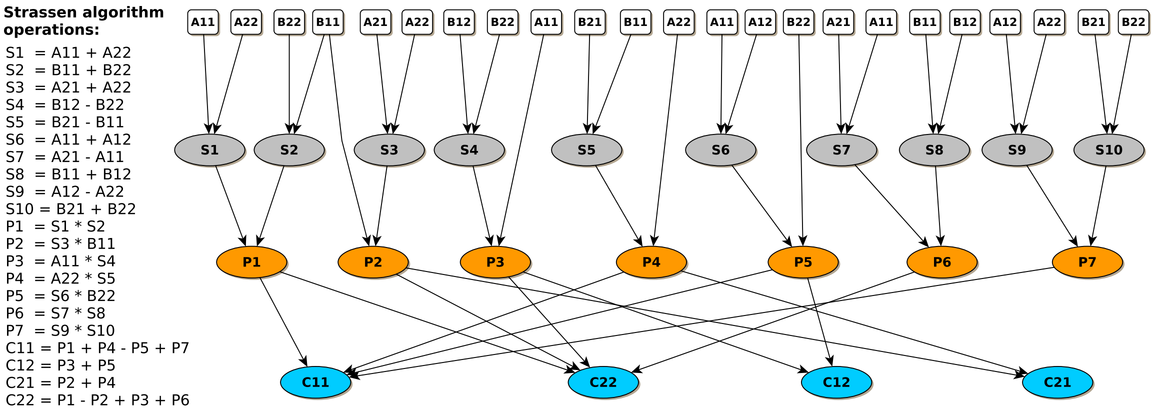


Figure 9-9: Dependencies instruction graph of the sequential Strassen's algorithm for the matrix multiplication $C = A \times B$.

par. degree	PF	MDF
1	500.2	505.3
8	62.4	62.9
16	31.3	31.5
24	20.8	23.3

Matrix size	seq	MDF	MDF+PF
2Kx2K	40.7	6.7	1.3
4Kx4K	341.8	56.2	10.8

Table 9.2: *Left:)* Execution time (in seconds) on the *Xeon* platform obtained by a *ParallelFor* implementation (PF) and by the *MDF* implementation of the same parallel loop. *Right:)* Strassen algorithm execution time (in seconds) on the *KNL* platform for different size of the matrices. Comparison between the sequential version (seq) the *MDF* and a second *MDF* implementation in which intermediate operations are executed using a *ParallelFor* (MDF+PF).

instruction is parallelized using a *ParallelFor* (MDF+PF in Table 9.2). In particular, we used seven Workers for the *MDF* pattern and 16 Workers for the *ParallelFor* pattern. The result obtained is a speedup of more than 30×, demonstrating that *pattern composition* may produce a performance boost without the need to devise more complex solutions. Indeed, in Section 9.4.2 we will see that the Strassen's algorithm can also be efficiently parallelized by using the *Divide&Conquer* pattern and a recursive algorithm.

A more complex test, is the Cholesky factorization algorithm (see also Sect 9.3.1). We consider dense matrices of single precision complex elements and elementary BLAS operations are executed using the Intel MKL library shipped with the In-

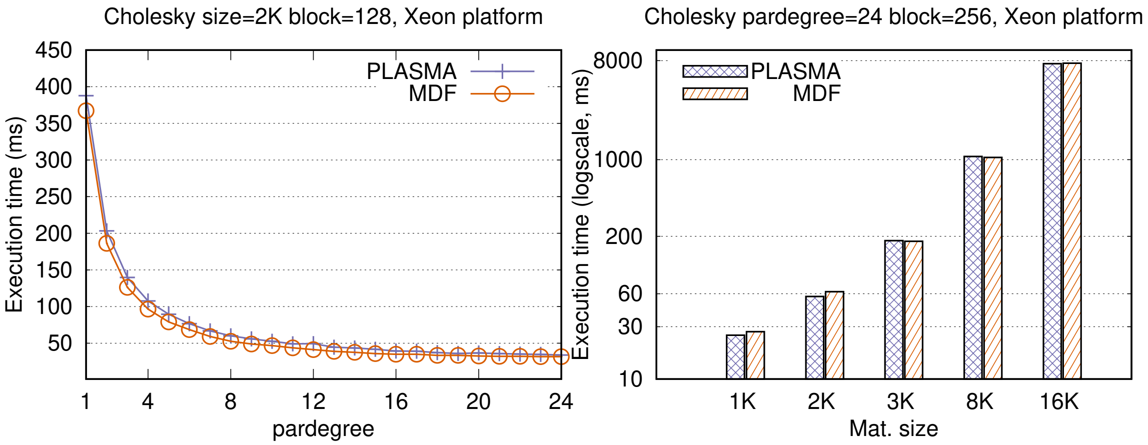
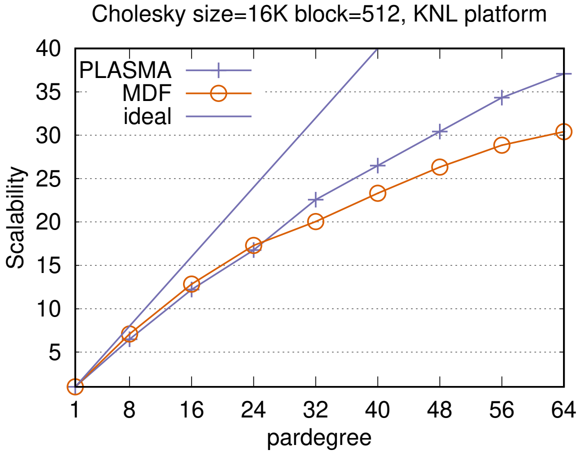


Figure 9-10: Left:) Cholesky factorisation execution time (milliseconds) for a matrix of 2K with a block size of 128 varying the parallelism degree. Right:) Best execution time varying the matrix size. The block size is fixed to 256 for all cases.

tel Parallel Studio XE 2017. In this case, we compare with a specialized numerical library optimized for multi-cores: the PLASMA library [71]. Specifically, we consider the static version of the PLASMA library which uses the left-looking Cholesky decomposition (the same version presented in Section 9.3.1) as it was determined to be the best for the static scheduler [142]. The results of the execution on the *Xeon* platform are reported in Figure 9-10. In the left-hand side of the figure is shown the execution time (in seconds) for a "small" matrix of size 2K ×2K and tiles of 128 × 128. On the right-hand side is shown the best execution time obtained using all machine cores, varying the size of the matrices. The performance of the *MDF* version is almost the same as the specialized PLASMA library using static scheduling.

We executed the same test on the *KNL* platform using a higher number of cores. The scalability for a matrix of $16K \times 16K$ is shown in the left-hand side of Figure 9-11. In this case the *MDF* version scales like the static PLASMA library up to 24 Workers and then it slows down obtaining a maximum scalability of 30.5 versus a scalability of 37 of the PLASMA version. The main motivation of this difference is that the static scheduler of the PLASMA version has a lower parallel overhead than the dynamic scheduler of the *MDF* pattern that with high parallelism degree has higher impact. The number of tasks scheduled by the *MDF* scheduler for different matrix sizes is



Matrix size	**N. of tasks**	
	blk=256	blk=512
1Kx1K	20	4
2Kx2K	120	20
4Kx4K	816	120
8Kx8K	5,984	816
16Kx16K	45,760	5,984

Figure 9-11: *Left:)* Scalability of the MDF version vs the PLASMA static version considering a matrix of 16K and a block of 512 × 512. *Right:)* Table reporting the number of tasks for different matrix sizes of the Cholesky factorization considering a blocks of 256 × 256 and 512 × 512.

reported in the right-hand side of Figure 9-11.

By considering all tests executed, the *MDF* pattern demonstrated good scalability. It also demonstrates that it is possible to build complex and efficient parallel patterns by properly assembly FastFlow building blocks and defining smart scheduling strategies by leveraging the *farm* Emitter.

An obvious extension of the MDF pattern implementation, consist in to use the *all-to-all* building block to parallelize the two sequential stages of the current version, i.e. task generator and scheduler, to improve the scalability with high parallelism degree.

9.4 Divide&Conquer

Divide & Conquer (DC) is a well-known recursive problem-solving strategy that divides the original problem into smaller sub-problems of the same type, each one solved recursively. Then, sub-problem solutions are properly combined to obtain the solution of the original problem. The idea is that finding the solution of sub-problems and combining their results is easier to do than trying to solve the entire problem directly. The strategy consists of two main steps applied at each level of recursion:

- a ***divide*** phase in which the problem is divided into sub-problems having the same type and a smaller size;

- a ***conquer*** phase in which the solutions of the sub-problems are merged to obtain the final solution of the initial problem.

A vast set of problems in different application domains can be solved with this method, typical examples are sorting algorithms (e.g., Mergesort). DC algorithms have been widely investigated given their intrinsic nature to be suitable for parallel computations. The executions on different sub-problems are usually independent and can be performed in parallel. In addition, DC algorithms tend to be *cache oblivious* [50], i.e. they can take advantage of both shared and private caches by having good spatial and/or temporal locality.

Despite their pronounced tendency to parallelism, the efficient parallel implementations of DC algorithms require a particular expertise in parallel programming and a good knowledge of parallel programming tools and frameworks to obtain the desired level of performance on today's multi-/many-core architectures. The FastFlow DC pattern [109], implemented by the C++ class `ff_DC`, aims to provide to non-expert parallel programmers a high-level, ready-to-use implementation of parallel DC algorithms on multi-core platforms, having performance comparable with hand-made parallelizations.

9.4.1 Pattern definition and implementation

To instantiate the `ff_DC` pattern the programmer has to provide the data type of the input problem and the type of the output result as template arguments. In the following we will refer to them as `ProblemType` and `ResultType` respectively. In the description, we consider them as different types, although they can coincide. To be utilized in the interface, the types must provide a default constructor. In addition, the programmer must provide the input object and the output object where the final result will be stored. These parameters are easily identifiable from the sequential code and, as indicated by Mattson et al. [241], they are sufficient to fully characterize the algorithm behavior:

- a `divide` function takes as input a problem and produces a set of sub-problems. It has the following interface:

  ```
  void divide(const ProblemType &p, std::vector<ProblemType> &subps);
  ```

  The function fills out the `subps` vector container passed by reference.

- a `condition` to test whether a problem is a base case problem:

  ```
  bool cond(const ProblemType &p);
  ```

- a `seq` function for solving the base case problem. It takes as input a problem and produces the corresponding result. Both of them are passed by reference:

  ```
  void base(const ProblemType &p, ResultType &res);
  ```

- a `combine` function that builds the result of a problem starting from the solution of its sub-problems:

  ```
  void combine(std::vector<ResultType>& subres, ResultType &res);
  ```

In a DC sequential algorithm, the recursion continues until the sub-problems can be solved directly. In a parallel program, it is typically more convenient to stop recursion at an optimal level of computation granularity and solve the problem sequentially. This may result in a better use of the memory hierarchy and lower the impact of the parallelization overhead. However, it may also limit the number of concurrent activities. The optimal *cutoff* size depends both on the specific problem and on target architecture as studied in recent research work [157]. In the current version of the FastFlow DC pattern, this value has to be provided by the programmer.

The different functional parameters must be provided as `std::function`, i.e. they can be any callable C++ object such as function pointer, lambda expression or function objects. An example of instantiation of the DC interface is shown in Code 26.

```
// functions aliases
using divide_f_t=std::function<void(const ProblemType&,
                                    std::vector<ProblemType>&)>;
using combine_f_t=std::function<void(std::vector<ResultType>&,
                                     ResultType&)>;
using base_f_t=std::function<void(const ProblemType&, ResultType&)>;
using cond_f_t=std::function<bool(const ProblemType&)>;
// D&C pattern constructor
template <typename ProblemType, typename ResultType>
ff_DC(const divide_f_t& divide, const combine_f_t& combine,
      const base_f_t& base, const cond_f_t& cond,
      const ProblemType& p, ResultType& res, int par_degree)
```

Code 26: DC pattern interface.

The programmer provides the reference to the starting problem (p), i.e. the input of the original algorithm, and a reference to the final result (res) where the result will be stored at the end of the parallel processing. Furthermore, the programmer may indicate also the desired number of parallel executors (par_degree) that by default is set to the number of available CPU cores. The call to the compute() method on the DC object will start the computation. Once returned, the result will be found in the res variable.

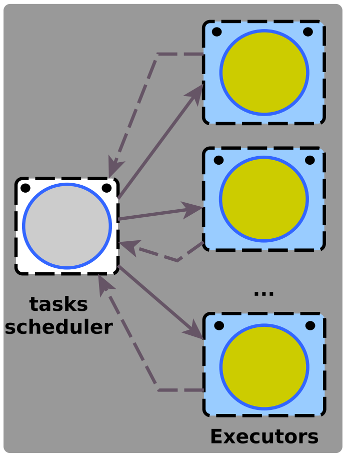
Mergesort examples. The advantage of using a high-level pattern-based approach is that all the parameters required to instantiate the pattern can be easily derived from the sequential algorithm. In addition, all the details concerning the parallel implementation are completely hidden from the programmer. As an exemplification, in Code 27 it is shown how to express the standard *Mergesort* algorithm using the DC pattern. To use the DC interface the programmer defines a Problem type that encapsulates the information needed to describe the problem. For the *Mergesorte* algorithm two iterators indicating the vector portion to be sorted, are sufficient. Besides, the same definition can be used as the Result type.

In the *divide* phase the problem of sorting an n-element sequence is divided into the problem of sorting two sub-sequences of $n/2$ elements (line 2 of Code 27). The *combine* phase is essentially managed by the merge function (lines from 12 to 20). In

```
1  struct Problem {  std::vector<long>::iterator left,right; };
2  void divide(const Problem &p,std::vector<Problem> &subps) {
3    std::vector<int>::iterator mid=p.left+(p.right-p.left)/2;
4    Problem a,b;
5    a.left=p.left;  a.right=mid;     subps.push_back(a);
6    b.left=mid;     b.right=p.right; subps.push_back(b);
7  }
8  void seq(const Problem &p, Result &ret) {
9    ret=p;  std::sort(ret.left,ret.right);
10 }
11 void merge(std::vector<Result>& res,Result& ret) {
12   long size=res[1].right-res[0].left;
13   std::vector<long> tmp(size);
14   std::vector<long>::iterator i=res[0].left, mid=res[0].right; j=mid;
15   for(long k=0;k<size;k++)     //merge in order
16     if(i<mid && (j>=res[1].right  *i<=*j))
17       tmp[k]=*i; i++;
18     else tmp[k]=*j; j++;
19   std::copy(tmp.begin(),tmp.end(),res[0].left);
20   ret.left =res[0].left;  ret.right=res[1].right;
21 }
22 bool cond(const Problem &p) { return (p.right-p.left<=CUT_OFF); }
23 int main() {
24   ... // load vector V that has to be sorted
25   Problem p(V.begin(),V.end());
26   Result res;
27   ff_DC<Problem,Result> dc(divide,merge,seq,cond,p,res,par_degree);
28   if (dc.run_and_wait_end()<0) return -1;
29 }
```

Code 27: DC implementation of the Mergesort algorithm.

this case, the programmer has to properly build the problem and the result data structures. Furthermore, in the parallel implementation, we have to consider the case that eventually the sub-problems generated by the divide function become small enough that they can be computed sequentially. Therefore, it could be more convenient to stop the recursion before reaching the base case of the sequential algorithm. This is captured in the cond function: the sequential version is used when the remaining size of the vector to be sorted has length smaller than a given *cutoff* parameter (e.g., 2, 000 elements) (line 22). The DC pattern is instantiated in the main function by using the ff_DC interface and setting also the number of threads (par_degree) to use in the run-time for the parallel execution (line 28).



```
void DC(const ProblemType &p,ResultType &ret) {
 if(!cond(op)) { //not the base case
   //divide
   std::vector<ProblemType> ps;
   divide(p,ps);
   std::vector<ResultType> res(ps.size());
   //conquer, recursive phase
   for(size_t i=0;i<ps.size();i++)
     DC(ps[i],res[i]);
   combine(res,ret);    //combine results
   return;
 }
 seq(p,ret); //base case
}
```

Figure 9-12: Left:) Building blocks used to implement the DC pattern. Right:) Recursive algorithm executed in parallel by the DC pattern.

Implementation with building blocks. The implementation of the DC pattern is based on the MDF implementation described in Section 9.3. In particular, it has been implemented as a dynamic macro data-flow interpreter processing direct acyclic graphs (DAG) of tasks generated at run-time. In this case, differently from the *MDF* case which implements a more general execution model, the algorithm that is parallelized with the DC pattern is fixed and it is the one sketched in the right-hand side of Figure 9-12. Tasks are created at each recursive call: independent calls can go through the recursion tree in parallel but they have to be synchronized before performing the *combine* phase, in order to be sure that all the partial results have been computed.

Its run-time is in charge of scheduling tasks to the processing units as they become available (fireable), i.e. all input data-dependencies are satisfied. For *Divide & Conquer* algorithms the DAG can be identified by considering the recursion tree. To guarantee the correctness, proper DAG dependencies are enforced among the graph nodes representing the sub-problems and the ones representing the partial results.

The management of the DAG is handled by a scheduler node which maintains only the part of the DAG that is needed for the computation. Task already computed are removed and memory deallocated. The building blocks structure of the DC pattern is a *farm* without Collector (see the left-hand side of Figure 9-12). The Emitter node is the scheduler node which manages the hash table implementing the DAG. Feedback channels between the Workers and the Emitter are used to notify the completion of operations assigned to the Worker.

9.4.2 Evaluation

In this section, we consider three different *Divide & Conquer* algorithms: the *Mergesort*, the *Quicksort* and the *Strassen's matrix multiplication* (simply called *Strassen*) algorithms. Besides being very well-known, these problems fully characterize the variety of possible situations that may occur in divide and conquer algorithms. The *Mergesort* algorithm is characterized by a *divide* phase with a negligible computation cost, while most of the running time is spent in the *combine* phase. The *Quicksort* algorithm is symmetric compared to Mergesort. The *combine* phase is totally absent and the entire work is essentially performed in the divide phase. Finally, in the *Strassen* algorithm both the divide and combine phases represent relatively coarse-grain computations. In the *Strassen* algorithm, differently from the other two algorithms, at each recursion step the problem is divided into seven sub-problems, rather than just two.

The `DC` pattern has been implemented by using a task-based approach in Intel TBB and OpenMP and also by using the FastFlow building blocks described in the previous sections. In the OpenMP and TBB versions, tasks are created at each recursive call. Independent calls can go through the recursion tree in parallel and they are synchronized before performing the *combine* phase to ensure that all the partial results have been computed. In the OpenMP implementation recursive calls of the algorithm described in Figure 9-12 have been annotated as `#pragma omp task`, the synchronization is implemented as `#pragma omp taskwait` to wait for task completion. In the TBB implementation, explicit calls to the low-level `tbb::task` class have

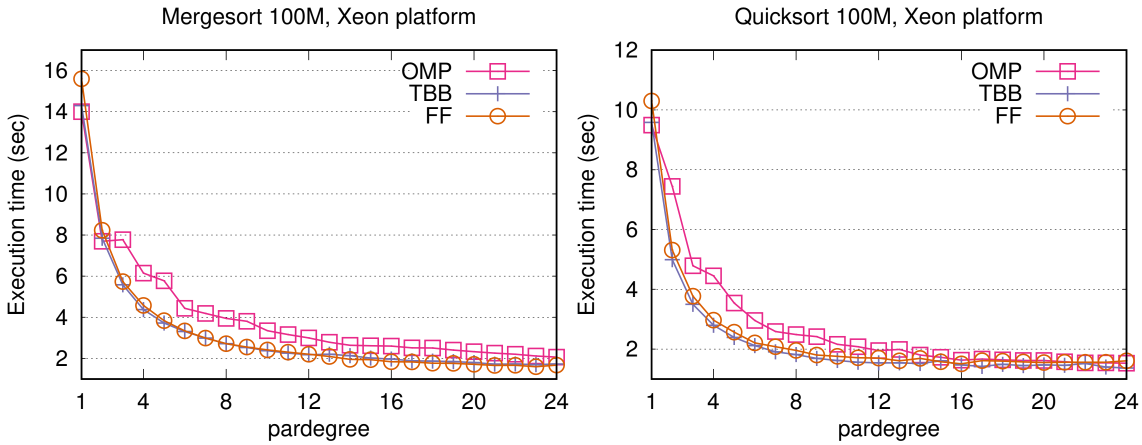


Figure 9-13: DC-based Mergesort and Quicksort algorithms implementation using TBB, OpenMP and FastFlow backends. Problem size: 100 million (100M) of integers elements.

been used to generate tasks and to synchronize their completion.

The programs are tested with different sizes of the input data: Mergesort and Quicksort are tested with arrays having a size equal to 10M, 20M, 50M and 100M integer elements. Strassen is tested using square dense matrices of double-precision elements having sizes equal to 1K×1K, 2K×2K, 4K×4K and 8K×8K.

The target platform used for the experiments is the *Xeon* server described in Table 6.1. The *Turboboost* and *Hyperthreading* facilities have been disabled. Concerning the libraries, for OpenMP we use the implementation provided with `gcc 6.4.0`, for the Intel TBB we used version `4.1`. All the measurements are performed multiple times and average values are shown: in general, the difference between the standard deviation and the average values reported is less than 3.5%.

Figure 9-13 shows the total execution time (or completion time) of the sorting algorithms with the biggest input data instances using different parallelism degrees. With parallelism degree equal to one, the FastFlow backend usually has a completion time higher than the OpenMP and TBB versions, however, it approaches the other two solutions as the parallelism degree increases. In the Mergesort case, FastFlow obtains the best time, while OpenMP obtains the worst. The TBB implementation is the best option for the Quicksort problem: we argue that this is due to the TBB task

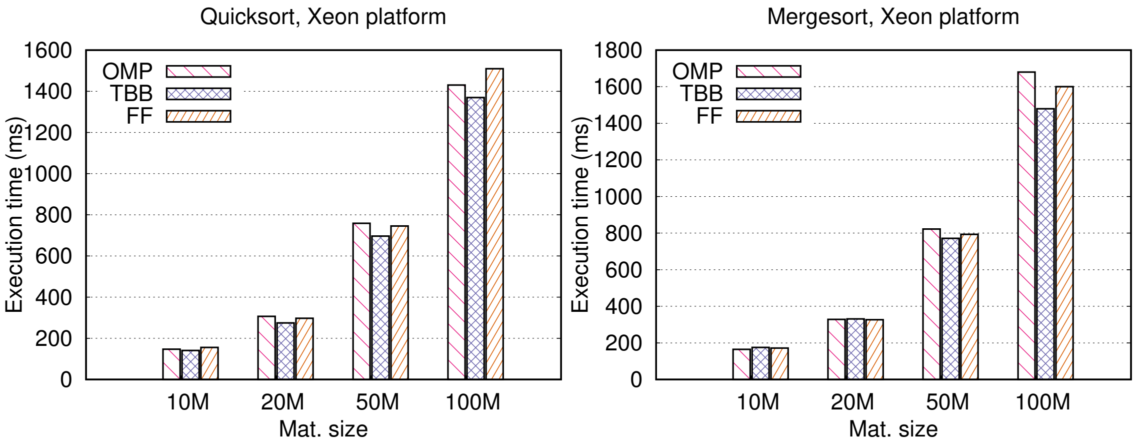


Figure 9-14: Best execution time for the DC pattern varying the problem size.

scheduler, which is known to be able to efficiently handle situations of unbalanced computations. This is exactly the situation characterizing Quicksort, where the divide phases can produce sub-problems with substantially different sizes. The *Mergesort* and *Quicksort* plots show a plateau when we use a parallelism degree equal or greater to $12 - 14$, which means that the performance does not steadily increase if we use more cores. The main reason for this behavior is due to the fact that the sorting problems are essentially memory-bound, hence the overall scalability of the parallel implementation using many cores is bounded by the memory bandwidth provided by the machine. Figure 9-14 shows the best completion time of the three RTSs by varying the problem size.

The completion time for the *Strassen* algorithm when using the biggest matrix size (8K) and by varying the problem size is shown in Figure 9-15. The TBB implementation of the *Strassen* algorithm exhibits the best completion time for large problem size, while OpenMP and FastFlow perform similarly, both slightly slower than the TBB version. The *Strassen* implementation provides a better scalability than the one obtained by the sorting algorithms. This is mainly due to the fact that the *Strassen* algorithm has a more balanced ratio between memory bandwidth requirements and CPU utilization.

In general, all the implementations of the *DC* pattern behave similarly. This demonstrates that the FastFlow RTS is able to provide similar performance figures to

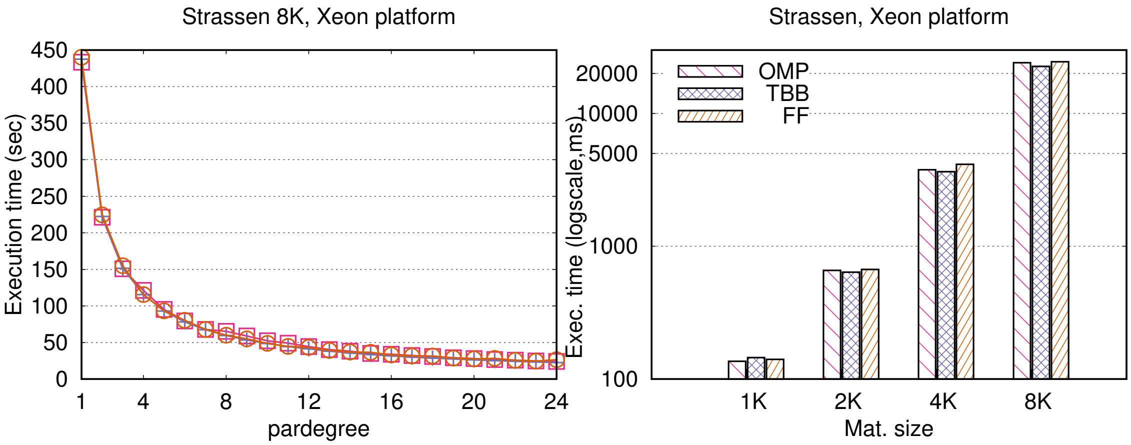


Figure 9-15: Left:) DC-based *Strassen* algorithm implementation for the maximum problem size 8K square matrix of double precision elements. Right:) Best execution time varying the problem size.

the ones offered by task-based run-times. However, the centralized scheduler used by the FastFlow implementation (the farm's Emitter) suffers scalability problems when the size of the problem and the number of available cores increases. As for the *MDF* pattern, the new *all-to-all* building block can be used to parallelize the centralization point, i.e. the *DC* scheduler.

9.5 Summary

In this chapter, we presented three *non-streaming* parallel patterns implemented on top of FastFlow parallel building blocks: the *ParallelFor*, the *Macro Data-Flow* and the *Divide & Conquer* patterns. We demonstrated that the FastFlow building blocks can be used to build not only streaming patterns but also efficient and ready-to-use task-based parallel patterns with good performance features.

The usability of the three patterns has been assessed using well-known algorithms and applications. The performance of the FastFlow implementations has been compared to state-of-the-art frameworks that offer either the same parallel pattern or specialized implementation of the kernel considered or providing lower-level mechanisms suitable to implement the pattern tested.

The experimental results demonstrate that the performance achieved by the three patterns presented is close to and in a few cases better than the ones offered by state-of-the-art implementations. The results obtained demonstrate the high flexibility of the FastFlow building blocks whose low parallel overhead and enhanced composability and customizability allow the user to implement new efficient parallel components with performance result as good as state-of-the-art parallel systems, but with implementation, flexibility, performance portability, and software engineering advantages.

We believe that the FastFlow programming model based on the proper composition of parallel building blocks presents a clear methodology for building parallel components capable of providing the programmer with the right level of abstraction to develop efficient and flexible solutions.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 10

Summary and Future Directions

With the broad diffusion of multi-core systems in almost all market segments, several industrial and research tools have been proposed for targeting efficient parallel software development on multi/many-cores (we discussed some of these recent proposals in Chapter 3). All of them, in different ways, are aiming at overcoming the limitations of the traditional low-level parallel programming models based on the mix of "threads & locks" mechanisms. Unfortunately, the majority of currently available parallel programming frameworks are either not high-level enough for the standard programmers or not flexible enough to be used in different application domains. Some proposals provide the user with mechanisms that we consider "low-level" abstractions such as pragma-based task annotations. Others promote programming models that allow implementing quite efficient programs, but they do not provide high-level abstractions for non-expert parallel programmers (e.g., Actor-based models). The ones that offer high-level parallel patterns capable of solving well-known classes of problems are mainly thought for parallel application developers, and typically they are not flexible enough to introduce *ad hoc* optimizations. Moreover, they are not easily extensible to allow the implementation of newer and more powerful high-level parallel components by RTS programmers.

In our vision, what is currently missing in the landscape of parallel programming frameworks and tools, is a common and widely accepted software layer of well-defined, highly-efficient and portable parallel components that can be used as common RTS to cover the usability needs and the performance requirements of different application domains. That would simplify parallel programming also for expert parallel programmers, allowing them to concentrate on developing suitable DSLs and ready-to-use parallel solutions instead of (re-)implementing new RTSs.

In this thesis, we try to fill this gap by proposing a new design and also some important new features of the FastFlow parallel programming library targeting shared-memory systems. The new design completes and strengthens the work we started in 2010 when we made available the initial version of the FastFlow library. Throughout this dissertation, we presented "the FastFlow way" to harness parallel programming and tackling the issues mentioned previously. The FastFlow library is based on the *structured parallel programming methodology* and its layered software design targets both RTS programmers and domain-expert applications programmers by offering them suitable parallel abstractions and sharing a common programming methodology. Our main contribution in this direction is the proposal of a *reduced set of sequential and parallel building blocks*, mimicking the RISC model of microprocessor architectures, that can be used both as fundamental components for building higher-level frameworks and also as useful parallel components that can be customized, connected with other building blocks, and nested for developing efficient parallel applications. In addition to building blocks, the FastFlow library provides a set of ready-to-use high-level parallel patterns built on top of building blocks that seamlessly integrate with them allowing non-expert parallel programmers to design a parallel version of their problem quickly and efficiently. FastFlow applications are defined according to the data-flow compositions of building blocks connected by bounded or unbounded FIFO channels carrying *streams* of data references. Shared-memory FastFlow channels are considered state-of-the-art and are one of the distinguishing features of the library we developed (communication channels are discussed in Chapter 6). Data-flow *streams* and *parallel building blocks* are first class citizens in the FastFlow library, and they are the two fundamental ingredients of the FastFlow parallel programming model.

The FastFlow library stands out over other solutions mainly for its demonstrated *extensibility* and *flexibility* in targeting different application domains and for offering both suitable abstractions for building high-level parallel patterns as well as efficient mechanisms for building RTSs. Such remarkable features, do not come at the expense of efficiency and performance, instead, as we demonstrated with several benchmarks and applications throughout this dissertation, they are aligned with (and in some cases better than) state-of-the-art mainstream parallel frameworks. The careful design of the library and its efficient low-level mechanisms (e.g., communication channels), together with a clear separation between the RTS code and the business logic code, allowed us to build the right mix of *flexibility*, *programmability*, and *performance*.

In light of the continuous evolution of multi/many-core platforms, *performance portability* is another crucial aspect we considered in this work. To this end, our contribution was to introduce to the FastFlow library a new software component called *concurrency graph transformer*, which offers the essential mechanisms to restructure and optimize the data-flow concurrency graph produced by patterns and building blocks compositions. Straightforward yet effective graph transformations are transparently and automatically provided to the FastFlow user through optimization flags (e.g., *farm fusion* and *farm combine* – presented in Chapter 8). Such transformations aim at reducing the number of graph's nodes and eliminating potential bottlenecks introduced by the compositions and nesting of building blocks having employing mediator nodes. The API provided by the transformation component allows the programmer to implement *ad hoc* building blocks transformations capable of accommodating the efficient execution of FastFlow applications on different target platforms.

Finally, throughout this dissertation, we extensively tested and assessed both low-level mechanisms and building blocks as well as the concurrency graph transformations by using simple kernels and notable parallel use-cases. We discussed and tested some high-level parallel patterns available in the library, i.e. *ParallelFor*, *Macro Data-Flow* and *Divide & Conquer*, which are built on top of the building block components we have developed. The experimental results demonstrate the high versatility and the good performance of the FastFlow parallel programming library.

In the following, we briefly recap the main results achieved:

- A thorough description of the FastFlow parallel library presenting its parallel programming model and its implementation design (Chapter 4).
- The definition and implementation of bounded and unbounded FIFO channels implemented by using Single-Produce Single-Consumer (SPSC) FIFO queues (Chapter 6).
- The implementation of both *blocking* and *non-blocking* concurrency control mechanisms for accessing the communication channels connecting two concurrent FastFlow nodes. The proposed protocol allows to dynamically switch the concurrency mode between passive waiting (*blocking*) and active waiting (*non-blocking*) of run-time threads. This is particularly relevant for long-running data streaming computations with wide input rate fluctuations (Chapter 6).
- The definition and implementation of a RISC-like set of parallel building blocks. Specifically, we have defined and implemented: a new parallel component called *all-to-all* modeling the *shuffle* communication pattern; the new *sequential node combiner* and a rich set of new composition and combining rules for the building blocks (Chapters 5, 7).
- A new FastFlow concurrency graph transformation software component. This component allows to statically (and in some cases automatically) introduce graph transformations capable of optimizing the FastFlow concurrency graph describing the application enabling *performance portability* and increasing resource usage efficiency (Chapter 8).
- The implementation of concurrency throttling mechanisms in the farm building block that enable the development of sophisticated policies to dynamically change the concurrency level of the *farm*'s Workers with the aim to increase either the sustained input rate or to reduce the power consumption by reducing the number of threads when the *non-blocking* concurrency control policy is used (Chapter 7).
- A thorough experimental validation of the proposed building blocks to sup-

port the efficient implementation of well-known parallel patterns, namely the *ParallelFor*, the *Divide&Conquer* and the *Macro-DataFlow* patterns (Chapters 6, 7, 8, 9).

Future research directions

There are several research directions worth investigating on the basis of what we have already developed in the FastFlow library. In the following, we briefly highlight some of them.

- Starting from the concurrency graph transformation mechanisms implemented in the new FastFlow concurrency graph transformation component, it would be interesting to implement a performance-model-driven static optimization component capable to automatically restructure and optimize (possibly guided by user's hints) the graph of nodes on the target platform. By using post-mortem data analysis of the execution metrics collected by the RTS on previous runs, the new component may derive suitable graph transformations with the aim of satisfying user-defined objectives such as for example to balance the service time among nodes or to reduce power consumption.
- An interesting research challenge consists in working on dynamic transformations of the concurrency graph describing the FastFlow application. The mechanisms we designed in the graph transformation software layer are thought to manage the graph statically before starting the parallel execution of nodes (i.e. at compile-time). However, the same graph transformations can be in principle introduced at run-time, not only as we have already done by enabling *concurrency throttling* of Workers in the *farm* building block (see Section 7.3.2), but also to merge and then restore stages and distinct parallel building blocks while the application is running. That would allow enlarging the optimization space: to better accommodate different performance needs of long-running applications, and to tackle dynamic resource variations typical of IoT and Fog environments.

- Developing new high-level parallel patterns targeting new application domains. Notably, there is a deluge of machine learning and deep learning algorithms used in many different applications. Studying both the workloads and the characteristics of these algorithms might lead to a definition of common components implemented by using the proposed building blocks and to "standardized" solutions to be provided to the applications programmer. However, even the implementation of fully general models of computation such as the *BSP model* targeting multi-cores is interesting and worth adding to the FastFlow list of high-level parallel patterns.

- In this dissertation, we concentrated on shared-memory systems. An interesting extension of this work would be to target distributed systems. We have already developed a distributed version of the FastFlow library, but that was a proof-of-concept, preliminary implementation and lots of work still have to be done. Primarily, it would be interesting to extend the building blocks parallel programming approach together with the concurrency graph transformations/optimizations policies to distributed memory systems with the aim of defining a unified framework offering the same programming model for both shared-memory as well as distributed-memory systems.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, Dec 1996.

[2] Sarita Vikram Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993. `http://rsim.cs.uiuc.edu/~sadve/Publications/thesis.pdf`.

[3] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. *SIGARCH Comput. Archit. News*, 17(3):396–406, April 1989.

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[6] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, July 2013.

[7] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *Int. J. Parallel Program.*, 42(6):1012–1031, December 2014.

[8] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *The Journal of Supercomputing*, Sep 2016. (In press).

[9] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in java. *Future Gener. Comput. Syst.*, 19(5):611–626, July 2003.

[10] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons for component autonomic management on grids. In *Making Grids Work: Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments 12-13 June 2007, Heraklion, Crete, Greece*, pages 3–15, Boston, MA, 2008. Springer.

[11] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in FastFlow. In Ioannis Caragiannis, Michael Alexander, Rosa Maria Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, pages 47–56, Berlin, Heidelberg, 2013. Springer.

[12] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool evolution: A parallel pattern for evolutionary and symbolic computing. *International Journal of Parallel Programming*, 44(3):531–551, Jun 2016.

[13] Marco Aldinucci, Massimo Coppola, and Marco Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 44–58. Fakultät für mathematik und informatik, Uni. Passau, Germany, May 1998.

[14] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.

[15] Marco Aldinucci and Marco Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems & Structures*, 33(3):179 – 192, 2007. Semantics and Cost Models for Cost Models for High-level Parallel Programming.

[16] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed amp; parallel application programming. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[17] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Skeletons for multi/many-core systems. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans Peters, and Thierry Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 265–272, Lyon, France, 2010. IOS press.

[18] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with FastFlow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, August 2011. Springer.

[19] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 662–673, Berlin, Heidelberg, 2012. Springer.

[20] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting heterogeneous architectures via macro data flow. *Parallel Processing Letters*, 22(2), 2012.

[21] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: High-Level and Efficient Streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. John Wiley & Sons, Inc, January 2017. accepted in 2012.

[22] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199. IEEE, February 2010.

[23] Marco Aldinucci, Guilherme Peretti Pezzi, Maurizio Drocco, Concetto Spampinato, and Massimo Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *The International Journal of High Performance Computing Applications*, 29(4):461–472, 2015.

[24] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. Decision Tree Building on Multi-core Using FastFlow. *Concurr. Comput. : Pract. Exper.*, 26(3):800–820, March 2014.

[25] Fernando Alexandre, Ricardo Marques, and Hervé Paulino. On the support of task-parallel algorithmic skeletons for multi-gpu computing. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 880–885, New York, NY, USA, 2014. ACM.

[26] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79, 1997.

[27] Akhtar Ali, Usman Dastgeer, and Christoph Kessler. Opencl for programming shared memory multicore cpus. In *Fifth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2012) at HiPEAC-2012, 23 January, Paris, France.* HiPEAC Network of Excellence, 2012.

[28] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[29] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[30] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[31] Henrique Andrade, Bura Gedik, and Deepak Turaga. *Fundamentals of Stream Processing.* Cambridge University Press, 2014. Cambridge Books.

[32] Joe Armstrong. The development of erlang. *SIGPLAN Not.*, 32(8):196–203, August 1997.

[33] S. Arnautov, P. Felber, C. Fetzer, and B. Trach. Ffq: A fast single-producer/multiple-consumer concurrent fifo queue. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 907–916, May 2017.

[34] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[35] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[36] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[37] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 851–862, Berlin, Heidelberg, 2009. Springer.

[38] B. Bacci, Marco Danelutto, S. Pelagatti, and M. Vanneschi. Skie: A heterogeneous environment for hpc applications. *Parallel Computing*, 25(13):1827 – 1852, 1999.

[39] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P^3 *L*: A structured high-level parallel language, and its structured support. *Concurrency - Practice and Experience*, 7(3):225–255, 1995.

[40] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[41] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, Oct 2015.

[42] Samy Al Bahra. Concurrency kit library, 2018. Last accessed June 2018: `http://concurrencykit.org/`.

[43] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

[44] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 761–770, Berlin, Heidelberg, 2005. Springer.

[45] Jost Berthold, Mischa Dieterle, and Rita Loogen. Implementing parallel google map-reduce in eden. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 990–1002, Berlin, Heidelberg, 2009. Springer.

[46] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[47] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[48] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 172–183, 2016.

[49] Richard S. Bird. An introduction to the theory of lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42, Berlin, Heidelberg, 1987. Springer.

[50] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[51] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[52] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[53] OpenMP Architecture Review Board. OpenMP Application Programming Interface – Version 4.5, 2015. Last accessed March 2018: `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[54] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995.

[55] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[56] Dan Bonachea. Gasnet specification v1.1. Technical Report UCB/CSD-02-1207, U.C.Berkeley, 2002. Newer versions available at https://gasnet.lbl.gov/.

[57] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.

[58] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[59] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[60] Andrea Bracciali, Marco Aldinucci, Murray Patterson, Tobias Marschall, Nadia Pisanti, Ivan Merelli, and Massimo Torquati. Pwhatshap: efficient haplotyping for future generation sequencing. *BMC Bioinformatics*, 17(11):342, Sep 2016.

[61] Stefan Breuer, Michael Steuwer, and Sergei Gorlatch. Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. In Armin Größlinger and Harald Köstler, editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 15–21, Vienna, Austria, January 2014.

[62] Antonio Brogi, Marco Danelutto, Daniele De Sensi, Ahmad Ibrahim, Jacopo Soldani, and Massimo Torquati. Analysing Multiple QoS Attributes in Parallel Design Patterns-Based Applications. *International Journal of Parallel Programming*, 46(1):81–100, Feb 2018.

[63] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. Cost-directed refactoring for parallel erlang programs. *Int. J. Parallel Program.*, 42(4):564–582, August 2014.

[64] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[65] D. Buono, Marco Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010.

[66] D. Buono, Marco Danelutto, S. Lametti, and Massimo Torquati. Parallel patterns for general purpose many-core. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 131–139, Feb 2013.

[67] D. Buono and G. Mencagli. Run-time mechanisms for fine-grained parallelism on network processors: The tilepro64 experience. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 55–64, July 2014.

[68] Daniele Buono. *Support models and cost models for structured parallel programming on multi and many cores.* PhD thesis, University of Pisa, 2014. `https://www.di.unipi.it/Documents/didattica/PhD/Thesis/2014/Buono.pdf`.

[69] Daniele Buono, Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. A Lightweight Run-Time Support For Fast Dense Linear Algebra on Multi-Core. In *Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014).* IASTED, ACTA press, February 2014.

[70] David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[71] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[72] Jan-Willem Buurlage, Tom Bannink, and Rob H. Bisseling. Bulk: A modern c++ interface for bulk-synchronous parallel programs. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018 Parallel Processing*, pages 519–532, Berlin, Heidelberg, 2018. Springer.

[73] Sonia Campa, Marco Danelutto, Mehdi Goli, Horacio González-Vélez, Alina Madalina Popescu, and Massimo Torquati. Parallel patterns for heterogeneous CPU/GPU architectures: Structured parallelism from cluster to cloud. *Future Generation Computer Systems*, 37:354 – 366, 2014.

[74] S. Campanoni, T. M. Jones, G. Holloway, G. Wei, and D. Brooks. Helix: Making the extraction of thread-level parallelism mainstream. *IEEE Micro*, 32(4):8–18, July 2012.

[75] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 1st edition, 2011.

[76] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53, Feb 2008.

[77] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, pages 72–81, Berlin, Heidelberg, 2007. Springer.

[78] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[79] David Castro, Kevin Hammond, Susmit Sarkar, and Yasir Alguwaifli. Automatically deriving cost models for structured parallel processes using hylomorphisms. *Future Gener. Comput. Syst.*, 79(P2):653–668, February 2018.

[80] Daniel Cederman, Anders Gidenstam, Phuong Ha, Hkan Sundell, Marina Papatriantafilou, and Philippas Tsigas. *Lock-Free Concurrent Data Structures*, chapter 3, pages 59–79. Wiley-Blackwell, 2017.

[81] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[82] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[83] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[84] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[85] Barbara Chapman. The multicore programming challenge. In *Advanced Parallel Processing Technologies: 7th International Symposium, APPT 2007 Guangzhou, China, November 22-23, 2007 Proceedings*, pages 3–3, Berlin, Heidelberg, 2007. Springer.

[86] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[87] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, pages 87–96, New York, NY, USA, Oct. 2013. ACM.

[88] Adriana E. Chis and Horacio González-Vélez. Design patterns and algorithmic skeletons: A brief concordance. In Joanna Kołodziej, Florin Pop, and Ciprian Dobre, editors, *Modeling and Simulation in HPC and Cloud Systems*, pages 45–56, Cham, 2018. Springer International Publishing.

[89] Märtin Christian. Post-dennard scaling and the final years of moore's law. Technical report, Hochschule Augsburg University of Applied Sciences, September 2014. Last accessed February 2018: http://www.hs-augsburg.de/Binaries/Binary20963/PostDennard.pdf.

[90] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain decomposition and skeleton programming with ocamlp31. *Parallel Comput.*, 32(7):539–550, September 2006.

[91] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[92] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 175–185, New York, NY, USA, 2011. ACM.

[93] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[94] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[95] R. Cole and O. Zajicek. The apram: Incorporating asynchrony into the pram model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 169–178, New York, NY, USA, 1989. ACM.

[96] ISO C++ Standard Committee. Working draft, standard for programming language C++, 2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[97] ISO C++ Standard Committee. Working draft, technical specification for C++ extensions for parallelism version 2 [n4725], 2018. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4725.html`.

[98] Intel Corporation. Intel xeon processor scalable family, 2018. Reference Number 336062-003. Last accessed September 2018: `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-scalable-datasheet-vol-1.pdf`.

[99] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[100] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[101] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[102] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computer Science & Engineering*, 5(1):46–55, January 1998.

[103] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.

[104] Griebler Dalvan. *Domain-Specific Language & Support Tools for High-Level Stream Parallelism.* PhD thesis, Pontifical Catholic University of Rio Grande do Sul, 2016. `https://gmap.pucrs.br/dalvan/papers/2016/thesis_dalvan_PUCRS_2016.pdf`.

[105] Griebler Dalvan, B. Hoffmann Renato, Danelutto Marco, and G. Fernandes Luiz. Higher-level parallelism abstractions for video applications with spar. In *Proceedings of International Parallel Computing Conference (ParCo)*, volume 32 of *Advances in Parallel Computing*, pages 698–707, Bologna, Italy, 2018. IOS Press.

[106] Marco Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[107] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, Massimo Torquati, Marco Aldinucci, and Peter Kilpatrick. The RePhrase Extended Pattern Set for Data Intensive Parallel Computing. *International Journal of Parallel Programming*, Nov 2017. (In Press).

[108] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. Data stream processing via code annotations. *The Journal of Supercomputing*, Jun 2016. (In Press).

[109] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. A divide-and-conquer parallel pattern implementation for multicores. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*, SEPS 2016, pages 10–19, New York, NY, USA, 2016. ACM.

[110] Marco Danelutto, Luca Deri, Daniele De Sensi, and Massimo Torquati. Deep packet inspection on commodity hardware using FastFlow. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans Peters, editors, *Proc. of 15th Inter. Parallel Computing Conference (ParCo)*, volume 25 of *Advances in Parallel Computing*, pages 92 – 99. IOS Press, 2013.

[111] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel Functional Programming with Skeletons: the OCamlP3L experiment. In *ACM Workshop on ML and its applications*, Baltimore, United States, September 1998. ACM.

[112] Marco Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and Massimo Torquati. Introducing Parallelism by Using REPARA C++11 Attributes. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 354–358, Feb 2016.

[113] Marco Danelutto, Peter Kilpatrick, Gabriele Mencagli, and Massimo Torquati. State access patterns in stream parallel computations. *The International Journal of High Performance Computing Applications*, 0(0):1094342017694134, 0. (In press).

[114] Marco Danelutto, T. De Matteis, D. De Sensi, and Massimo Torquati. Evaluating concurrency throttling and thread packing on smt multicores. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 219–223, March 2017.

[115] Marco Danelutto, D. De Sensi, and Massimo Torquati. Energy driven adaptivity in stream parallel computations. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 103–110, March 2015.

[116] Marco Danelutto, Daniele De Sensi, and Massimo Torquati. A power-aware, self-adaptive macro data flow framework. *Parallel Processing Letters*, 27(1):1–20, 2017.

[117] Marco Danelutto and Massimiliano Stigliani. Skelib : Parallel programming with skeletons in C. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings.*, pages 1175–1184, 2000.

[118] Marco Danelutto and Massimo Torquati. A RISC Building Block Set for Structured Parallel Programming. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 46–50, Feb 2013.

[119] Marco Danelutto and Massimo Torquati. Loop parallelism: A new skeleton perspective on data parallel patterns. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 52–59, Feb 2014.

[120] Marco Danelutto and Massimo Torquati. Increasing efficiency in parallel programming teaching. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 306–310, March 2018.

[121] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming. *Procedia Computer Science*, 80:1519 – 1530, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[122] J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*, pages 160–169, Sept 1993.

[123] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. *SIGPLAN Not.*, 30(8):19–28, August 1995.

[124] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for gpu-based systems. *Int. J. Parallel Program.*, 44(3):506–530, June 2016.

[125] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the skepu skeleton programming library. In Chenggang Wu and Albert Cohen, editors, *Advanced Parallel Processing Technologies Workshop (APPT)*, pages 170–183, Berlin, Heidelberg, 2013. Springer.

[126] Tiziano De Matteis and Gabriele Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13:1–13:12, 2016.

[127] Tiziano De Matteis and Gabriele Mencagli. Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *International Journal of Parallel Programming*, pages 1–20, 2016.

[128] Daniele De Sensi. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, University of Pisa, 2010. `https://www.di.unipi.it/Documents/didattica/PhD/Thesis/2018//DeSensiPhDThesis.pdf`.

[129] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.*, 14(4):33:1–33:26, October 2017.

[130] Daniele De Sensi, Peter Kilpatrick, and Massimo Torquati. State-aware concurrency throttling. In *Proceedings of International Parallel Computing Conference (ParCo 2017)*, Advances in Parallel Computing, pages 201–210, Bologna, Italy, 2018. IOS Press.

[131] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, dec 2016.

[132] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. Mammut: High-level management of system knobs and sensors. *SoftwareX*, 6:150 – 154, 2017.

[133] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[134] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[135] David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, and J. Daniel Garca. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, pages e4175–n/a, 2017. e4175 cpe.4175.

[136] David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, and J. Daniel Garca. Paving the way towards high-level parallel pattern interfaces for data stream processing. *Future Generation Computer Systems*, 87:228 – 241, 2018.

[137] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[138] J. Diaz, C. Muoz-Caro, and A. Nio. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, Aug 2012.

[139] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[140] Manuel F. Dolz, David DelRioAstorga, Javier Fernndez, Massimo Torquati, Jos Daniel Garca, Flix Garca-Carballeira, and Marco Danelutto. Enabling semantics to improve detection of data races and misuses of lock-free data structures. *Concurrency and Computation: Practice and Experience*, 29(15), 2017. (In press).

[141] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[142] Joseph Dorris, Jakub Kurzak, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Task-based cholesky decomposition on knights corner using openmp. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, pages 544–562, Cham, 2016. Springer International Publishing.

[143] Maurizio Drocco. *Parallel Programming with Global Asynchronous Memory: Models,C++ APIs and Implementations*. PhD thesis, University of Torino, 2017. `https://zenodo.org/record/1037585#.W6JQuRRoSkA`.

[144] Alejandro Duran, Eduard Ayguade, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[145] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, New York, NY, USA, 2005.

[146] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[147] Steffen Ernsting and Herbert Kuchen. Algorithmic Skeletons for Multi-core, multi-GPU Systems and Clusters. *Int. J. High Perform. Comput. Netw.*, 7(2):129–138, April 2012.

[148] August Ernstsson, Lu Li, and Christoph Kessler. Skepu2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, Feb 2018.

[149] The eSkel group. The edinburgh skeleton library, August 2005. Last accessed April 2018: `http://homepages.inf.ed.ac.uk/mic/eSkel/`.

[150] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, May 2012.

[151] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[152] Babak Falsafi. Server Architecture for the Post-Moore Era, 2017. Keynote talk at the 2nd Workshop "Hot Topics in Data Centers" (HotDC), October 2017, Hefei, China. Last accessed February 2018: `https://parsa.epfl.ch/~falsafi/talks/HotDC2017.pdf`.

[153] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *USENIX ATC 16)*, pages 393–406, Denver, CO, 2016. USENIX Association.

[154] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, March 2012.

[155] Paulo Ferrão, Hélder Maques, and Hervé Paulino. Stream processing on hybrid cpu/intel xeon phi systems. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018 Parallel Processing*, pages 796–810, Berlin, Heidelberg, 2018. Springer.

[156] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[157] Alcides Fonseca and Bruno Cabral. Evaluation of runtime cut-off approaches for parallel programs. In *Proceedings of the 12th International Meeting on High Performance Computing for Computational Science (VECPAR 2016)*. Springer, 2016. (In Press).

[158] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[159] Message Passing Interface Forum. Mpi: A Message-Passing Interface Standard – Version 3.1, June 2015. Last accessed March 2018: `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[160] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[161] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[162] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[163] Leonardo Gazzarri and Marco Danelutto. A tool to support FastFlow program design. In *Proceedings of the International Conference on Parallel Computing (ParCo 2017)*, Advances in Parallel Computing, pages 687–697, Bologna, Italy, 2018. IOS Press.

[164] Buğra Gedik. Generic windowing support for extensible stream processing systems. *Softw. Pract. Exper.*, 44(9):1105–1128, September 2014.

[165] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[166] P. B. Gibbons. A more practical pram model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM.

[167] Roberto Giorgi, Rosa M. Badia, Franois Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R. Gao, Arne Garbade, Rahul Gayatri, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Kolia, Joshua Landwehr, Nhat Minh L, Feng Li, Mikel Lujn, Avi Mendelson, Laurent Morin, Nacho Navarro, Tomasz Patejko, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Ian Watson, Sebastian Weis, Stphane Zuckerman, and Mateo Valero. Teraflux: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976 – 990, 2014.

[168] Mehdi Goli. *Autonomic behavioural framework for structural parallelism over heterogeneous multi-core systems.* PhD thesis, Robert Gordon University, Aberdeen, UK, 2015. `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.702194`.

[169] Mehdi Goli and Horacio González-Vélez. Formalised composition and interaction for heterogeneous structured parallelism. *International Journal of Parallel Programming*, 46(1):120–151, Feb 2018.

[170] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, November 2010.

[171] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures.* PhD thesis, Massachusetts Institute of Technology, 2010. AAI0822890.

[172] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30(5):291–303, October 2002.

[173] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 492–499, April 1999.

[174] Sergei Gorlatch and Murray Cole. Parallel skeletons. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1417–1422, Boston, MA, 2011. Springer.

[175] Peter Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7, 2011. Last accessed March 2018: `https://www.cl.cam.ac.uk/~rdm34/big.LITTLE.pdf`.

[176] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory.* Oxford University Press, Inc., New York, NY, USA, 1995.

[177] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, 2017.

[178] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz G. Fernandes. Stream parallelism with ordered data constraints on multi-core systems. *The Journal of Supercomputing*, Jul 2018. (First online version).

[179] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, Cambridge, MA, USA, 2nd edition, 1999.

[180] Aaftab Munshi (Khronos OpenCL Working Group). The OpenCL Specification version 1.1, 2010. Last accessed February 2018: `https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf`.

[181] A. Gupta and V. Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19(3):234 – 244, 1993.

[182] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[183] M. Haidl and S. Gorlatch. PACXX: Towards a unified programming model for programming accelerators using C++14. In *2014 LLVM Compiler Infrastructure in HPC*, pages 1–11, Nov 2014.

[184] Michael Haidl and Sergei Gorlatch. High-level programming for many-cores using C++14 and the STL. *International Journal of Parallel Programming*, 46(1):23–41, Feb 2018.

[185] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. PACXXv2 + RV: An llvm-based portable high-performance programming model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, pages 7:1–7:12, New York, NY, USA, 2017. ACM.

[186] Kevin Hammond. Functional programming and the "megacore" era. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 1–1, New York, NY, USA, 2014. ACM.

[187] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects: 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, pages 218–236, Berlin, Heidelberg, 2013. Springer.

[188] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July 2011.

[189] Danny Hendler and Nir Shavit. Work dealing. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 164–172, New York, NY, USA, 2002. ACM.

[190] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[191] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd Int. Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[192] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 18–25, Dec 1991.

[193] L. Higham and J. Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, pages 56–63, Dec 1997.

[194] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Computing*, 24(14):1947 – 1980, 1998.

[195] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[196] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[197] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70, Feb 2011.

[198] iMatix Corporation. Zeromq distributed messaging, 2018. Last accessed August 2018: `http://zeromq.org/`.

[199] Typesafe Inc. Akka library. Last accessed June 2018: `http://akka.io`.

[200] David Mulnix (Intel). Intel Xeon Processor Scalable Family Technical Overview, September 2017. Last accessed February 2018: `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`.

[201] Noman Javed and Frédéric Loulergue. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In Yong Dou, Ralf Gruber, and Josef M. Joller, editors, *Advanced Parallel Processing Technologies*, pages 436–451, Berlin, Heidelberg, 2009. Springer.

[202] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[203] James Jeffers, James Reinders, and Avinash Soldani. *Intel Xeon Phi Coprocessor High Performance Programming – Knights Landing Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2st edition, 2016.

[204] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[205] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[206] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language version 1.10 revision 59, 2004. Last accessed March 2018: `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.1.10.pdf`.

[207] Christoph Kessler and Jörg Keller. Models for parallel computing: Review and perspective, 2007. PARS-Mitteilungen 24: 13-29, ISSN 0177-0454, GI/ITG PARS.

[208] Kathleen Knobe. Ease of use with concurrent collections (cnc). In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 17–17, Berkeley, CA, USA, 2009. USENIX Association.

[209] J. Korinth, D. d. l. Chevallerie, and A. Koch. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 195–198, May 2015.

[210] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM.

[211] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. *SIGPLAN Not.*, 30(6):196–204, June 1995.

[212] Herbert Kuchen. A skeleton library. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 620–629, Berlin, Heidelberg, 2002. Springer.

[213] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM.

[214] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[215] Silvia Lametti. *Structured Parallel Programming and Cache Coherence in Multicore Architectures*. PhD thesis, University of Pisa, 2015. `https://www.di.unipi.it/Documents/didattica/PhD/Thesis/2015/Lametti.pdf`.

[216] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[217] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2014.

[218] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001. Last accessed May 2018: `https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`.

[219] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[220] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[221] Jongwuk Lee and Seung won Hwang. Scalable skyline computation using a balanced pivot selection technique. *Information Systems*, 39:1 – 21, 2014.

[222] Joeffrey Legaux, Frdric Loulergue, and Sylvain Jubertie. Osl: An algorithmic skeleton library with exceptions. *Procedia Computer Science*, 18:260 – 269, 2013. 2013 International Conference on Computational Science.

[223] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[224] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, March 2005.

[225] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[226] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[227] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk synchronous parallel ml: Modular implementation and performance prediction. In Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science – ICCS 2005*, pages 1046–1054, Berlin, Heidelberg, 2005. Springer.

[228] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: A toolkit for content-based similarity search of feature-rich data. *SIGOPS Oper. Syst. Rev.*, 40(4):317–330, April 2006.

[229] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 61–70 vol.2, Jan 1995.

[230] S. Maleki, Y. Gao, M. J. Garzarn, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, Oct 2011.

[231] Danelutto Marco and Mencagli Gabriele. D2.7:final report on patterns and relationship with general design patterns, September 2017. Last accessed June 2018: `https://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d2-7.pdf`.

[232] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 896–907, New York, NY, USA, 2003. ACM.

[233] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: Better strategies for parallel haskell. *SIGPLAN Not.*, 45(11):91–102, September 2010.

[234] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. Algorithmic skeleton framework for the orchestration of gpu computations. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 874–885, Berlin, Heidelberg, 2013. Springer.

[235] M. Martineau, S. McIntosh-Smith, and W. Gaudin. Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 338–347, May 2016.

[236] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. An evaluation of emerging many-core parallel programming models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, pages 1–10, New York, NY, USA, 2016. ACM.

[237] Torquati Massimo, Danelutto Marco, and Aldinucci Marco. D6.3: Dynamic runtimes with auto-tuning capabilities, February 2016. Document ID ICT-609666-D6.3. Last accessed June 2018: `http://repara-project.eu/wp-content/uploads/2016/04/ICT-609666-6.3.pdf`.

[238] Kiminori Matsuzaki and Kento Emoto. Lessons from implementing the bicgstab method with sketo library. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 15–24, New York, NY, USA, 2010. ACM.

[239] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A fusion-embedded skeleton library. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, pages 644–653, Berlin, Heidelberg, 2004. Springer.

[240] Tiziano De Matteis. *Parallel Patterns for Adaptive Data Stream Processing.* PhD thesis, Computer Science Dept., University of Pisa, 2016. `https://www.di.unipi.it/Documents/didattica/PhD/Thesis/2016//DeMatteis.pdf`.

[241] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, first edition, 2004.

[242] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[243] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[244] Paul E McKenney. Is parallel programming hard, and, if so, what can you do about it?(v2017. 01.02 a). *arXiv preprint arXiv:1701.00854*, 2017.

[245] Memcached. Memcached, May 2018. Last accessed May 2018: `https://memcached.org`.

[246] G. Mencagli, Massimo Torquati, Marco Danelutto, and T. De Matteis. Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2608–2624, Sept 2017.

[247] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Generation Computer Systems*, 79:862 – 877, 2018.

[248] Gabriele Mencagli, Massimo Torquati, Fabio Lucattini, Salvatore Cuomo, and Marco Aldinucci. Harnessing sliding-window execution semantics for parallel stream processing. *Journal of Parallel and Distributed Computing*, 116:74 – 88, 2018. Towards the Internet of Data: Applications, Opportunities and Future Challenges.

[249] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[250] Claudia Misale. *PiCo: A Domain-Specific Language for Data Analytics Pipelines.* PhD thesis, University of Torino, 2017. `https://iris.unito.it/handle/2318/1633743`.

[251] Claudia Misale, Maurizio Drocco, Guy Tremblay, Alberto R. Martinelli, and Marco Aldinucci. Pico: High-performance data analytics pipelines in modern c++. *Future Generation Computer Systems*, 87:392 – 403, 2018.

[252] Mark Moir and Nir Shavit. Concurrent data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook Of Data Structures And Applications*, Chapman & Hall/Crc Computer and Information Science Series, chapter 47. Chapman & Hall/CRC, 2004.

[253] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sept 2006.

[254] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 852–863, May 2011.

[255] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[256] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[257] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[258] NVIDA. Nvidia Tesla P100, 2018. Last accessed March 2018: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[259] NVIDA. Parallel Thread Execution ISA v6.1, 2018. Last accessed February 2018: `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.1.pdf`.

[260] NVIDIA. Jetson TK1 Development Kit Specification, 2014. Last accessed March 2018: `http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/JTK1_DevKit_Specification.pdf`.

[261] NVIDIA). CUDA C Programming Guide, 2018. Last accessed August 2018: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[262] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, April 2014.

[263] OpenACC-Standard.org. The OpenACC Application Programming Interface – Version 2.6, 2018. Last accessed March 2018: `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf`.

[264] Daniel Orozco, Elkin Garcia, Rishi Khan, Kelly Livingston, and Guang R. Gao. Toward high-throughput algorithms on many-core architectures. *ACM Trans. Archit. Code Optim.*, 8(4):49:1–49:21, January 2012.

[265] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krueger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 2007.

[266] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[267] Susanna Pelagatti. *Methodologies and tools for structured highly parallel computing.* PhD thesis, University of Pisa, 1991. PhD Thesis.

[268] Susanna Pelagatti. *Structured development of parallel programs*, volume 102. Taylor & Francis Abington, 1998.

[269] Simon J Pennycook, Jason D Sewall, and VW Lee. A metric for performance portability. *arXiv preprint arXiv:1611.07409*, 2016.

[270] S.J. Pennycook, J.D. Sewall, and V.W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 2017.

[271] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151, Sept 2008.

[272] F. Petrini, Wu chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network (QsNet): high-performance clustering technology. In *HOT 9 Interconnects. Symposium on High Performance Interconnects*, pages 125–130, Aug 2001.

[273] Greg Pfister. An Introduction to the InfiniBand Architecture. In Rajkumar Buyya, Toni Cortes, and Hai Jin, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 42. IEEE, 2002.

[274] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, August 2009.

[275] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.

[276] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 651–665, New York, NY, USA, 2016. ACM.

[277] The LLVM Project. Clang: a C language family frontend for LLVM, 2010. Last accessed March 2018: `http://clang.llvm.org`.

[278] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Feb 2007.

[279] Thomas Rauber and Gudula Rnger. *Parallel Programming: For Multicore and Cluster Systems.* Springer Publishing Company, Incorporated, 2nd edition, 2013.

[280] RedisLab. Redis, May 2018. Last accessed May 2018: `https://redis.io`.

[281] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015.

[282] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[283] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

[284] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[285] Lee Howes (Khronos OpenCL Working Group SYCL subgroup) Ronan Keryell, Maria Rovatsou. Sycl specification version 1.2.1, 2017. Last accessed March 2018: `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`.

[286] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM.

[287] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[288] Karl Rupp. 42 Years of Microprocessor Trend Data, 2018. Last accessed June 2018: `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`.

[289] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[290] Tobias Schüle. Embedded multicore building blocks parallel programming made easy, 2015. Embedded World 2015 Conference. Last accessed June 2018: `https://embb.io/downloads/EMBB_Embedded-World_2015.pdf`.

[291] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, Oct 2015.

[292] A. Secco, I. Uddin, G. P. Pezzi, and M. Torquati. Message passing on infiniband rdma for parallel run-time supports. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 130–137, Feb 2014.

[293] Daniele De Sensi, Tiziano De Matteis, and Marco Danelutto. Simplifying self-adaptive and power-aware computing with nornir. *Future Generation Computer Systems*, 87:136 – 151, 2018.

[294] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb 1997.

[295] K. Siddique, Z. Akhtar, E. J. Yoon, Y. Jeong, D. Dasgupta, and Y. Kim. Apache hama: An emerging bulk synchronous parallel computing framework for big data applications. *IEEE Access*, 4:8879–8887, 2016.

[296] Mahendra Pratap Singh and Manoj Kumar Jain. Article: Evolution of processor architecture in mobile phones. *International Journal of Computer Applications*, 90(4):34–39, March 2014.

[297] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[298] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core.* MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[299] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011.

[300] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[301] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[302] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[303] Michel Steuwer and Sergei Gorlatch. Skelcl: Enhancing opencl for high-level programming of multi-gpu systems. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 258–272, Berlin, Heidelberg, 2013. Springer.

[304] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA, 2011. IEEE Computer Society.

[305] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[306] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 52–78. Springer, 2013.

[307] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005. Last accessed February 2018: http://www.gotw.ca/publications/concurrency-ddj.htm.

[308] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M. Voelker. Sora: High-performance software radio using general-purpose multi-core processors. *Commun. ACM*, 54(1):99–107, January 2011.

[309] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[310] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer.

[311] M. Torquati, G. Mencagli, M. Drocco, M. Aldinucci, T. De Matteis, and M. Danelutto. On Dynamic Memory Allocation in Sliding-Window Parallel Patterns for Streaming Analytics. *The Journal of Supercomputing*, Sep 2017. (In press).

[312] Massimo Torquati. Single-Producer/Single-Consumer Queues on Shared Cache Multi-Core Systems. Technical Report TR-10-20, Università di Pisa, Dipartimento di Informatica, December 2010.

[313] Massimo Torquati, T. Menga, T. De Matteis, D. De Sensi, and G. Mencagli. Reducing Message Latency and CPU Utilization in the CAF Actor Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 145–153, March 2018.

[314] Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, Marco Aldinucci, and Marco Danelutto. Power-aware pipelining with automatic concurrency control. *Concurrency and Computation: Practice and Experience*, 2018. (In Press).

[315] J. Torrellas, H. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, Jun 1994.

[316] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[317] Marco Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.

[318] Marco Vanneschi. *High performance computing. Parallel processing models and architectures.* Pisa University Press, 2014.

[319] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.

[320] Željko Vrba, Håvard Espeland, Pål Halvorsen, and Carsten Griwodz. Limits of work-stealing scheduling. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing: 14th Int. Workshop, JSSPP 2009, Rome, Italy, May 29, 2009. Revised Papers*, pages 280–299, Berlin, Heidelberg, 2009. Springer.

[321] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

[322] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 1st edition, 2009.

[323] Sandra Wienke, Christian Terboven, James C. Beyer, and Matthias S. Müller. A pattern-based comparison of openacc and openmp for accelerator computing. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, pages 812–823, Cham, 2014. Springer International Publishing.

[324] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[325] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[326] Xilinx. Vivado High-Level Synthesis, May 2018. Last accessed May 2018: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[327] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 16:1–16:13, New York, NY, USA, 2016. ACM.

[328] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer.

[329] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. PhD thesis, EECS Department, University of California, Berkeley, Feb 2014.

[330] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[331] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[332] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[333] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008.