UNIVERSITY OF TORINO

DOCTORAL SCHOOL ON SCIENCE
AND HIGH TECHNOLOGY

COMPUTER SCIENCE DEPARTMENT

DOCTORAL THESIS

Parallel Programming with Global Asynchronous Memory: Models, C++ APIs and Implementations

Author:
Maurizio DROCCO

Supervisor:
Prof. Marco ALDINUCCI

Cycle XXIX

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

"Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will."

Leslie Lamport

UNIVERSITY OF TORINO

Abstract

Computer Science Department

Doctor of Philosophy

Parallel Programming with Global Asynchronous Memory: Models, C++ APIs and Implementations

by Maurizio DROCCO

In the realm of High Performance Computing (HPC), message passing has been the programming paradigm of choice for over twenty years. The durable MPI (Message Passing Interface) standard, with send/receive communication, broadcast, gather/scatter, and reduction collectives is still used to construct parallel programs where each communication is orchestrated by the developer-based precise knowledge of data distribution and overheads; collective communications simplify the orchestration but might induce excessive synchronization.

Early attempts to bring shared-memory programming model—with its programming advantages—to distributed computing, referred as the Distributed Shared Memory (DSM) model, faded away; one of the main issue was to combine performance and programmability with the memory consistency model. The recently proposed Partitioned Global Address Space (PGAS) model is a modern revamp of DSM that exposes data placement to enable optimizations based on locality, but it still addresses (simple) data-parallelism only and it relies on expensive sharing protocols.

We advocate an alternative programming model for distributed computing based on a Global Asynchronous Memory (GAM), aiming to *avoid* coherency and consistency problems rather than solving them. We materialize GAM by designing and implementing a *distributed smart pointers* library, inspired by C++ smart pointers. In this model, public and private pointers (resembling C++ shared and unique pointers, respectively) are moved around instead of messages (i.e., data), thus alleviating the user from the burden of minimizing transfers. On top of smart pointers, we propose a high-level C++ template library for writing applications in terms of dataflow-like networks, namely GAM nets, consisting of stateful processors exchanging pointers in fully asynchronous fashion.

We demonstrate the validity of the proposed approach, from the expressiveness perspective, by showing how GAM nets can be exploited to implement both standalone applications and higher-level parallel programming models, such as data and task parallelism. As for the performance perspective, preliminary experiments show both close-to-ideal scalability and negligible overhead with respect to state-of-the-art benchmark implementations. For instance, the GAM implementation of a high-quality video restoration filter sustains a 100 fps throughput over 70%-noisy high-quality video streams on a 4-node cluster of Graphics Processing Units (GPUs), with minimal programming effort.

Acknowledgements

Walking all the way through a meaningful PhD program is an act of pure masochism. In addition to "the willingness to fail all the time" (cit. J. Backus), one needs to get comfortable with the constant feeling that your understanding on anything is wrong or, at best, incomplete. Therefore, getting some proper teachers along the way is essential to survive.

First, I would like to thank professor Tremblay for his inestimable help with this thesis. Guy, I hope to have eventually acquired, by osmosis, a small fraction of your impressive ability in building visions over any research topic.Furthermore, if native English readers can read this thesis without getting hurt to death, it is due to your corrections and lessons, that I hope to have learned, at least partially.

I would like to thank the reviewers for dedicating part of their time, impressive knowledge, and experience to improve this thesis. I perceived all your suggestions and remarks as a privilege, that made me feel "standing on the shoulders of giants".

I owe special thanks to professor Aldinucci for being much more than my supervisor during the last seven years. Marco, in your shop I built my special glasses for seeing both computer science and the world under a different light. You assisted me throughout the whole spectrum of my endless torments, from false sharing to teenage heartaches. Last but not least, it is your fault if I felt in love with the two most dangerous and beautiful beasts I met so far: parallel computing and Claudia.

My walk would not have been that funny without my colleagues, as masochist as me. Guys, together we went through a bunch of though challenges, from squeezing twenty hours of working plus five hours hanging around into a twenty-four day, to mastering the art of collecting receipts, which were never enough. I really enjoyed each step with you all.

Back to where my walk began, I owe a mix of thanks and apologies to my relatives. Dad, I spent whole days trying to mimic your artisan's hands, I steadily failed but at least I learned the concept of a job well done. Mom, thank you for showing me how honesty, humbleness, and solidarity can be more that beautiful words. Emi, you have always been my favorite brother, I have to thank you at least for all the advices and recipes that made me a slightly better cooker along the way.

Besides playing with doughs, I tend to be a theoretician, in the sense I like to spend hours searching for the right way to describe things, and of course I got accustomed to the feeling of not being completely happy with the descriptions I build. Walking with you, Claudia, has always been a different story: every day you gift me the exact colors that I need to define the perfect world. I wish I had a fraction of your wiseness and I will never ever understand "perché fra i tanti, bella, che hai colpito, ti sei gettata addosso proprio a me" (cit. F. Guccini).

Contents

Chapter 1

Introduction

In parallel computing, the *message-passing* and *shared-memory* programming models have been influencing programming at all levels of abstraction, from hardware to application design. These models have been traditionally considered a dichotomy, often mapped onto another dichotomy: *scalability* versus *productivity*.

In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various *synchronization* mechanisms may be used to control access to the shared memory. An advantage of this model from the programmer's viewpoint is that the notion of data distribution is absent, so there is no need to explicitly specify the mapping between data structures and Processing Elements (PEs). Thus, program development is generally simpler. For this to happen, the shared-memory model must exhibit a "natural" behavior to the programmer, in the sense that it should enable the programmer to design parallel code without data races and exhibiting a deterministic behavior. This requirement amounts to having a reasonably strong *memory consistency model*. This implies guaranteeing some global order of access to the shared memory from many PEs, which requires (at some level in the system stack) synchronizations and results into latency of memory accesses. Here, the strength of shared memory becomes a weakness already at a moderate scale: a strong memory consistency model over a shared memory is hardly scalable, whereas a weak model makes programming counterintuitive, eventually deteriorating the major strength of the shared-memory model, that is, simplicity of programming. The Distributed Shared Memory (DSM) hype cycle, which entered in the "trough of disillusionment" in the late nineties, is a paradigmatic example of the complexity of designing both an efficient and easy to use shared-memory model. More recently, the Partitioned Global Address Space (PGAS) approach, that couples shared memory with data parallelism, revitalized the research on DSM models. Is distributed shared memory entering the Gartner's hype cycle [82] "slope of enlightenment"?

By contrast, in the message-passing model, tasks use their own local memory during computation. Multiple tasks can reside on the same physical machine as well as across an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfers usually require cooperative operations to be performed by each process (e.g., a send operation must have a matching receive operation). More recently, one-sided communications gained progressive interest because they admit zero-copy hardware-accelerated data movements and promote loose orchestration of tasks.

The durable MPI (Message Passing Interface) standard, with send/receive communication, broadcast, gather/scatter, and reduction collectives is still used to construct parallel programs composed of tens to hundreds of thousands of communicating processes. Each communication is orchestrated by the developer-based precise knowledge of code and overhead; collective communications simplify the orchestration but induce excessive synchrony due to barriers and global synchronization induced by blocking collective operations. An MPI application is really a monolith where each single process may become a bottleneck or a single point of failure. This programming model is effective for highly regular data parallel kernels but difficult to exploit for other patterns, where dealing with data locality is difficult and beyond the control of the average user.

The endeavor for extreme scale computing, also catered by the Big Data analytics hype, certainly revamped the research on high-level parallel programming models and languages. Skeletal approach [51] evolved and, eventually, went mainstream with Intel TBB and Google MapReduce. Task-based approaches are ruling the game of large scale distributed programming; we shall discuss some of them. Notwithstanding, the run-time support of all these programming environment eventually relies on a either a message-passing or a shared-memory layer. We mainly focused on this level of abstraction, at the hardware-software interface.

In this thesis, we propose to overcome the aforementioned dichotomy, advocating a hybrid model where data is shared and data races are addressed by way of asynchronous message passing. On this cornerstone, we build an entire stack of programming models of increasing abstraction level.

The main objective of the present thesis is to define a novel programming approach for distributed (possibly large) heterogeneous platforms, covering the complete software stack from low-level runtime systems to application programming. The main contributions of the thesis directly match the tiers of the proposed stack, which is sketched in Fig. 1.1. Each tier of the stack defines either a novel programming model or the evolution of an existing one.

1.1 Results and Contributions

Global Asynchronous Memory

At the bottom layer of the stack in Fig. 1.1, the Global Asynchronous Memory (GAM) model consists in a set of *executors*[1] that share a global address space and that can make memory operations on the global addresses. A global address refers to a GAM slot, i.e., a binary object with either *public* or *private* attribute, according to the associated access capabilities. A single GAM slot is not distributed. A public slot can be accessed by any executor via `load` or `store` operations, although it cannot be updated once a value has been stored into it—i.e., GAM public slots are *single-assignment*. A public GAM slot can be replicated (cached) across different executors. Conversely, a private slot can be accessed via `load` and `store` operations,

[1] In this thesis, the *executor* is an abstract concept, that should be thought as a mere syntactic component in the GAM model formalization.

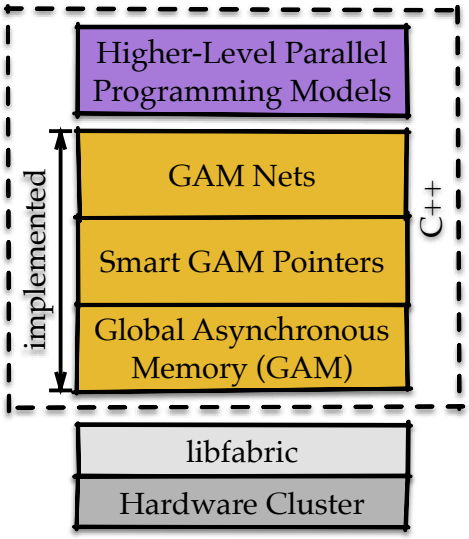


FIGURE 1.1: Contributions of this thesis.

but only by its owner, that is, in any given moment, by the executor owning exclusive access capability over the slot. Ownership can be atomically passed from one executor to another. Overall, the GAM obeys the *sequential consistency* memory model.

Inspired by the FastFlow programming model [17], the GAM exploits both shared memory and message passing programming models. In this new model, the executors synchronize with each other by passing messages that are *capabilities*, i.e., global memory references enriched with slot access attributes.

As discussed in Chapter 3, the GAM model avoids the problem of efficiently enforcing an adequately strong consistency model rather than solving it (as happened for DSMs). Also, it clearly differentiates itself from the PGAS approach since the partitioning, i.e., the data parallel commitment, is not deeply embedded in its definition. Nevertheless, as discussed in Chapter 6, GAM can be used to support a PGAS approach.

Smart GAM Pointers

The GAM fits in the modern C++ realm by way of *Smart GAM Pointers*. They are a novel abstraction that extend C++11 smart pointers toward distributed memory platforms, such as multicomputers and clusters, with or without hardware-assisted remote memory access. As in the traditional shared-memory setting of C++11, smart pointers aim at facilitating the development of correct code. Smart pointers prevent most situations of *memory leaks* by making the memory deallocation automatic. In this setting, they support dynamic GAM slot allocation and make their deallocation automatic: a memory slot controlled by a smart pointer is automatically destroyed when the last (or only) owner is destroyed. Smart pointers also eliminate *dangling pointers* by postponing destruction until an object is no longer in use.

As discussed in Chapter 4, along the same line as the distinction between shared and unique pointers in C++, we propose two classes of pointers, namely *public* and *private*. Public pointers resemble shared pointers, in the sense that different *copies* of a public pointer, distributed among the executors, share the control over the underlying (public) memory slot. Symmetrically, private pointers resemble unique pointers, in the sense that each pointer has exclusive control over the underlying (private) memory slot. Private pointers are designed to be strictly coupled with move semantics.

Smart GAM Pointers are implemented by a C++11 template library. It distinguishes itself from other libraries by its attempt to minimize the conceptual gap against shared-memory programming in C++. Being simple and orthodox should be one of its strengths. We envision distributed computing as a mainstream feature of standard C++. For this, complex aspects of sharing (such as data races) should be tamed through familiar concepts by way of graceful abstractions, which do not require an exceptional expertise to be used to build distributed C++ applications. The forthcoming C++ standard releases are clearly moving toward a full embedding of parallel transformations into the Standard Template Library (STL). This effort is currently limited to (cache-coherent) shared-memory programming model. This thesis aims forward and attempts to build on the embedding of distributed memory in mainstream C++.

GAM Nets

The programming model exposed by the Smart GAM Pointer tier envisions a collection of executors that exchange C++ smart pointers to (shared) data in a global, sequentially consistent distributed memory. According to the memory model, data can be accessed without data races (and without serialization). This crucially depends on capabilities and their movement among executors. Observe that this happens in a purely message passing style—indirect addressing of smart pointers is (deliberately) not supported by the GAM.

As discussed in Chapter 5, GAM Nets provide executors with message passing mechanisms to exchange pointers. In principle, they can be substituted with any message passing machinery, including MPI, as explored in preliminary work [74]. GAM Nets distinguish themselves from MPI along three main directions: 1) Computations and communications are clearly distinguished; GAM Nets provide a communication layer for executors rather than collective operations; executors have no role in supporting collective communications. 2) Communications are described by compoundable parametric patterns that can be statically evaluated for correctness and performance, e.g., mapped onto a fixed degree network such as a 2D torus. 3) GAM Nets realize fully asynchronous collective communications by design; no barriers, global or group synchronizations are necessary.

Furthermore, the GAM Nets tier represents a novel design of the "Distributed FastFlow" model [8], which addresses a number of weaknesses of the previous design (e.g., dynamic memory allocation). Moreover, the dependency on the ZeroMQ library has been replaced with a design based on the more general OFI (OpenFabrics Interface) framework.

Higher-Level Parallel Programming Models

GAM Nets naturally target stream-parallel programming. However, as widely demonstrated in the literature, stream parallelism can be fruitfully exploited to implement other models. For instance, several frameworks for high-level parallel programming, such as OpenMP [116], FastFlow [17], and Flink [79], exploit stream parallelism for implementing data-parallel operations.

To exemplify the depicted approach, in Chapter 6, we discuss how GAM nets can be exploited to implement a library of containers with data-parallel operations, along the same line as the transformations—and their parallel variants—recently introduced in C++. We also discuss how GAM programs can be regarded as runtime systems in the context of task-based processing, targeting distributed platforms. Examples of recently proposed task-based frameworks include OCR [101] and HPX [90].

In addition to the considered examples, other models could be easily implemented in terms of GAM programs, from low-level PGAS languages (e.g., UPC [77], Chapel [45], X10 [46]) to high-level Domain-Specific Languages (DSLs).

1.2 Limitations

The main limitation of this thesis is the lack of a full-fledged experimental evaluation on large-scale High Performance Computing (HPC) clusters. Instead, we focused on *variety* and *heterogeneity* when selecting the platforms for the evaluation. For the former aspect, we considered three different networking hardware, namely, Ethernet, InfiniBand, and A3Cube RONNIEE; moreover, we considered both commodity and high-end workstations as cluster nodes. For the latter aspect, we considered the cluster-of-Graphics Processing Units (GPUs) architecture, including the case of multiple GPUs per cluster node, in addition to plain multi-core nodes.

Another limitation, from the programmability perspective, is the lack of any mechanism for automatizing the serialization (and de-serialization) of objects to be sent over the network by the GAM runtime. In this thesis, we assume that each value stored in the GAM memory has a representation in memory that allows the value to be copied by simply replicating its byte sequence. However, it would be easy to provide some mechanism for (semi-)automatic serialization, on top of a serialization library such as Boost.Serialize [39] or Google Protobuf [98].

Moreover, although we present smart global pointers by analogy with C++ smart pointers (see Chap. 4), we do not provide *weak* global pointers, that express non-owning references in their shared-memory counterparts. Consequently, in its current version, the proposed implementation does not support circular references of smart global pointers.

Finally, the current implementation does not exploit Remote Memory Access (RMA) primitives, though they are provided by most networking hardware nowadays. To this aim, we envision an improved implementation, in which we allocate GAM memory from RMA-capable regions of GAM executors' address space, to enable implementing GAM accesses as direct calls to RMA primitives.

1.3 List of Papers

In this section, I report the complete list of my publications, in reverse chronological order. Although the listed publications do not directly refer to the contributions included in the present thesis, most of them acted as either inspiration, preliminary study, or use case for the GAM stack, as we detail in the following.

We organize the publications along two dimensions. In Sect. 1.3.1, we categorize them based on the targeted *topic*, whereas in Sect. 1.3.2, we categorize them based on the publication type (i.e., journals, conferences, and others).

1.3.1 Publications by Topic

High Performance Tools for Big Data

In recent years, an increasingly inter-connected ecosystem of heterogeneous devices has been producing larger volumes and variety of digital data. Those large volumes of dynamically changing data ought to be processed, synthesized, and eventually turned into knowledge. High-velocity data brings high value, especially to volatile business processes, mission-critical tasks, and scientific grand challenges. Some of this data lose their operational value within a short time frame, some other are simply too much to be stored. Because of this, data science is destined (sooner or later) to meet high-performance computing beyond parallel processing of batches on the file system.

In this context, we recently proposed a novel C++14-based DSL based on a layered dataflow model for processing data collections, called PiCo (*Pi*peline *Co*mposition). The main entity of this programming model 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 completely data management and focusing only on operations, which are represented by Pipeline stages (see Sec. 7.1.4). Designing and coding an application with PiCo is easier than in Spark or Flink, and, on tested cases, PiCo is faster (sometime much faster) because it is designed according to HPC best practices. PiCo is built on top of the FastFlow library, and currently runs on shared-memory platforms. The GAM Nets tier, being fully compatible with FastFlow, will make it possible to easily port PiCo to distributed platforms. An extensive performance benchmarking of PiCo on top GAM Nets is among our future work. Related papers are the following: *J1, J2, C1, C2, C3, O1*.

High-level Programming Models

Parallel programming models are concerned with abstractions for parallel computing. The value of a programming model is primarily related with its expressiveness (for a given target class of algorithms) and its performance. The implementation of a parallel programming model can take the form of a library invoked from a sequential language, i.e., as an extension to an existing language, or as an entirely new language (even if the language does not necessarily define a new programming model).

We approached parallel programming convinced of the need to raise the level of abstraction with respect to the state of the art. By definition, the raison d'être for high-performance computing is... high performance. But peak Floating Point Operations Per Second (FLOPS) count is not the only measure to evaluate the impact of these technologies. Human productivity, total cost, time-to-market, reliability, energy consumption, etc., are equally, if not more important factors for any industrial follow-up. To date, attempts to develop high-level programming abstractions, tools, and environments for HPC have mostly failed. Suitable abstractions, however, are the keys to induce an industrial impact. Over the past twenty years, Web service programmers have built and embraced an ecosystem of libraries, scripting languages, software services, and tools that allowed them to create complex systems while hiding most of the underlying details of networks and computer systems. Their focus is on composition, abstraction, rapid deployment, software scaling, and human productivity. In sharp contrast, in the realm of HPC, message passing has remained the programming paradigm of choice for over twenty years.

Following the experience of *algorithmic skeletons* [51, 53], and using FastFlow as a laboratory [17], we experimented a number of parallel patterns to simplify the development of applications running on heterogeneous platforms, including multi-core platforms attached to multiple GPUs. Probably the most relevant is the hybrid Stencil-Reduce pattern, presented at Nvidia GPU Technology Conference (GTC) 2014 in San Jose, CA, USA. This approach (ported to OpenCL) significantly evolved during the "REPARA" EU FP7 to become the run-time engine of a high-level parallel programming approach based on "parallelization hints" over a standard C++ language. Differently from the mainstream OpenMP approach that uses compiler directives, in REPARA, parallelization directives are introduced as C++ attributes, which are part of the C++ standard rather than an extension to the language. A similar spirit motivated part of the present work, advocating the evolution of the C++ language to distributed systems rather than building on it yet another extension. Related papers are the following: *J3*, *J5*, *J7*, *C4*, *C5*, *C9*, *C12*, *C16*.

Applications

In parallel computing, benchmarks and applications are the standard tools to validate the research work. Parallel programming models are often concerned with several aspects of software engineering, such as scalability, portability, programmability, dependability. For this, we often invested on (complex) applications rather than kernels, as they make it possible to have a thorough evaluation of the proposed technique. Some of those applications have became research topic per se:

- *Image and Video Restoration*. This application is based on a novel algorithm for edge-preserving image processing based on variational analysis. The algorithm exhibits an exceptionally high restoration capability for images with high level of noise (e.g., 90% of impulsive noise, see Fig. 7.15) at the price of a high computational cost. The application can exploit both stream and data parallelism and is compute-bound. Also, it can greatly benefit from GPU acceleration.

The application is also one of the use cases of this thesis (see Sec. 7.1.1). Related papers are the following: *J3, J5, J8, C5, C10, C13, O2, O3*.

- *Next-generation Sequencing (NGS)*. Sequencing costs are rapidly decreasing because of new massively parallel sequencing technologies. The number of sequences available during the last years has experienced an amazing growth, making most of the existing analysis tools obsolete. In testing parallel programming models, and specifically FastFlow, a tool for the analysis of *Chromosome Conformation Capture data* has been proposed (called *NuChart-II)*. Interestingly enough from the parallel computing viewpoint, this analysis requires to represent genomic information as a very large graph. The construction of the graph from raw DNA data is an irregular and memory-bound problem that is challenging to be coded in a scalable way. Its implementation on top of GAM Nets is among our possible future work. Related papers are the following: *J4, J6, C6, C7, C8*.

- *Systems Biology*. The CWC (Calculus of Wrapped Compartments) Systems Biology Simulator provides stochastic simulation of biological systems, which is a popular technique in Bioinformatics, in particular for its superior ability to describe transient and multi-stable behaviors of biological systems. However, stochastic simulation is computationally expensive, and the cost increases if the whole simulation-analysis workflow is considered. The efficient design of such a workflow is an interesting problem of parallel computing, since the frequency and size of data moved across the workflow strictly depend on the required accuracy. Indeed, it involves the merging of results from different simulation instances and possibly their statistical description or mining with data reduction techniques. The implementation of this application on top of GAM Nets is discussed in Sect. 7.1.3. Related papers are the following: *J7, J8, J9, J10, C12, C14, C15, C16, C17, C18*.

1.3.2 Publications by Type

Journal Papers

(J1) C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017

(J2) 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. *Journal of Supercomputing*, 2017. To appear

(J3) M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016

(J4) F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, M. Torquati, and M. Aldinucci. NuChart-II: the road to a fast and scalable tool for Hi-C data analysis. *International Journal of High Performance Computing Applications (IJHPCA)*, pages 1–16, 2016

(J5) M. Aldinucci, G. Peretti Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Applications*, 29(4):461–472, 2015

(J6) I. Merelli, F. Tordini, M. Drocco, M. Aldinucci, P. Liò, and L. Milanesi. Integrating multi-omic features exploiting Chromosome Conformation Capture data. *Frontiers in Genetics*, 6(40), 2015

(J7) M. Aldinucci, C. Calcagno, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, M. Torquati, and A. Troina. On designing multicore-aware simulators for systems biology endowed with on-line statistics. *BioMed Research International*, 2014

(J8) M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, C. Misale, C. Calcagno, and M. Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, 15(5):798–813, 2014

(J9) M. Coppo, F. Damiani, M. Drocco, E. Grassi, E. Sciacca, S. Spinella, and A. Troina. Simulation techniques for the calculus of wrapped compartments. *Theoretical Computer Science*, 431:75–95, 2012

(J10) M. Coppo, F. Damiani, M. Drocco, E. Grassi, M. Guether, and A. Troina. Modelling ammonium transporters in arbuscular mycorrhiza symbiosis. *Transactions on Computational Systems Biology (TCS)*, 6575(13):85–109, 2011

Conference Papers

(C1) M. Drocco, C. Misale, G. Tremblay, and M. Aldinucci. A formal semantics for data analytics pipelines. Technical report, Computer Science Department, University of Torino, May 2017

(C2) C. Misale, M. Drocco, G. Tremblay, and M. Aldinucci. Pico: a novel approach to stream data analytics. In *Euro-Par 2017 Workshops - Autonomic Solutions for Parallel and Distributed Data Stream Processing (AutoDasp)*, Santiago de Compostela, Spain, 2017. (Accepted)

(C3) C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. In *Proc. of HLPP2016: Intl. Workshop on High-Level Parallel Programming*, pages 1–19, Muenster, Germany, July 2016. arXiv.org

(C4) M. Drocco, C. Misale, and M. Aldinucci. A cluster-as-accelerator approach for SPMD-free data parallelism. In *Proc. of Intl. Euromicro PDP 2016: Parallel Distributed and network-based Processing*, pages 350–353, Crete, Greece, 2016. IEEE

(C5) M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, G. Peretti Pezzi, and M. Torquati. The loop-of-stencil-reduce paradigm. In *Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara)*, pages 172–177, Helsinki, Finland, Aug. 2015. IEEE

(C6) F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci. Parallel exploration of the nuclear chromosome conformation with NuChart-II. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*. IEEE, Mar. 2015

(C7) M. Drocco, C. Misale, G. Peretti Pezzi, F. Tordini, and M. Aldinucci. Memory-optimised parallel processing of Hi-C data. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, pages 1–8. IEEE, Mar. 2015

(C8) F. Tordini, M. Drocco, I. Merelli, L. Milanesi, P. Liò, and M. Aldinucci. NuChart-II: a graph-based approach for the analysis and interpretation of Hi-C data. In C. D. Serio, P. Liò, A. Nonis, and R. Tagliaferri, editors, *Computational Intelligence Methods for Bioinformatics and Biostatistics - 11th International Meeting, CIBB 2014, Cambridge, UK, June 26-28, 2014, Revised Selected Papers*, volume 8623 of *LNCS*, pages 298–311, Cambridge, UK, 2015. Springer

(C9) M. Aldinucci, M. Drocco, G. Peretti Pezzi, C. Misale, F. Tordini, and M. Torquati. Exercising high-level parallel programming on streams: a systems biology use case. In *Proc. of the 2014 IEEE 34th Intl. Conference on Distributed Computing Systems Workshops (ICDCS)*, Madrid, Spain, 2014. IEEE

(C10) M. Aldinucci, G. Peretti Pezzi, M. Drocco, F. Tordini, P. Kilpatrick, and M. Torquati. Parallel video denoising on heterogeneous platforms. In *Proc. of Intl. Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*, 2014

(C11) M. Drocco, M. Aldinucci, and M. Torquati. A dynamic memory allocator for heterogeneous platforms. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, Fiuggi, Italy, 2014. HiPEAC

(C12) M. Aldinucci, F. Tordini, M. Drocco, M. Torquati, and M. Coppo. Parallel stochastic simulators in system biology: the evolution of the species. In *Proc. of Intl. Euromicro PDP 2013: Parallel Distributed and network-based Processing*, Belfast, Nothern Ireland, U.K., Feb. 2013. IEEE

(C13) M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, and S. Palazzo. A parallel edge preserving algorithm for salt and pepper image denoising. In K. Djemal, M. Deriche, W. Puech, and O. N. Ucan, editors, *Proc. of 2nd Intl. Conference on Image Processing Theory Tools and Applications (IPTA)*, pages 97–102, Istambul, Turkey, Oct. 2012. IEEE

(C14) M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, M. Torquati, and A. Troina. On parallelizing on-line statistics for stochastic biological simulations. In *Euro-Par 2011 Workshops, Proc. of the 2st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*, volume 7156 of *LNCS*, pages 3–12, Bordeaux, France, 2012. Springer

(C15) C. Calcagno, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, and A. Troina. Modelling spatial interactions in the arbuscular mycorrhizal symbiosis using the calculus of wrapped compartments. In

I. Petre and E. P. de Vink, editors, *Proc. of Third International Workshop on Computational Models for Cell Processes (CompMod)*, volume 67 of *EPTCS*, pages 3–18, Aachen, Germany, Sept. 2011

(C16) M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, M. Torquati, and A. Troina. On designing multicore-aware simulators for biological systems. In Y. Cotronis, M. Danelutto, and G. A. Papadopoulos, editors, *Proc. of Intl. Euromicro PDP 2011: Parallel Distributed and network-based Processing*, pages 318–325, Ayia Napa, Cyprus, Feb. 2011. IEEE

(C17) M. Coppo, F. Damiani, M. Drocco, E. Grassi, E. Sciacca, S. Spinella, and A. Troina. Hybrid calculus of wrapped compartments. In G. Ciobanu and M. Koutny, editors, *Proc. of 4th Workshop on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC)*, volume 40 of *EPTCS*, pages 102–120, Jena, Germany, Aug. 2010

(C18) M. Coppo, F. Damiani, M. Drocco, E. Grassi, and A. Troina. Stochastic calculus of wrapped compartments. In A. D. Pierro and G. Norman, editors, *Proc. of the 8th Workshop on Quantitative Aspects of Programming Languages (QAPL)*, volume 28 of *EPTCS*, pages 82–98, Paphos, Cyprus, Mar. 2010

Other (Technical Reports, Posters, Talks, etc.)

(O1) M. Drocco, C. Misale, G. Tremblay, and M. Aldinucci. A formal semantics for data analytics pipelines. Technical report, Computer Science Department, University of Torino, May 2017

(O2) M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato. Fastflow: Combining pattern-level abstraction and efficiency in GPGPUs. In *GPU Technology Conference (GTC 2014)*, San Jose, CA, USA, Mar. 2014

(O3) M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato. An overview of fastflow: Combining pattern-level abstraction and efficiency in GPGPUs. In *GPU Technology Conference (GTC 2014)*, San Jose, CA, USA, Mar. 2014

Funding

This work has been partially supported by the Italian Ministry of Education and Research (MIUR), by the EU-H2020 RIA project "Toreador" (no. 688797), the EU-H2020 RIA project "Rephrase" (no. 644235), the EU-FP7 STREP project "REPARA" (no. 609666), the EU-FP7 STREP project "Paraphrase" (no. 288570), and the 2015-2016 IBM Ph.D. Scholarship program. Experimentation was made possible thanks to the A3Cube Inc. donation of RONNIEE networking boards, and to Compagnia di SanPaolo for the donation of the OCCAM heterogeneous cluster.

Chapter 2

Background

In this chapter, we provide the background that will help the reader go through the arguments of the thesis. In particular, we discuss concepts and approaches from the literature on parallel computing, which acted as both inspiration and basis for comparison with respect to the contributions of the thesis.

This chapter proceeds as follows. In Sect. 2.1, we review the most common parallel computing platforms. In Sect. 2.2, we review state-of-the-art approaches for parallel programming. In Sect. 2.3, we introduce and discuss parallel memory models. Finally, in Sect. 2.4, we present the frameworks and libraries that we actually exploited for concretizing the contributions of the thesis.

2.1 Parallel Computing Platforms

Computing hardware has evolved to sustain the demand for high-end performance along two basic ways. On the one hand, the increase in clock frequency and the exploitation of instruction-level parallelism boosted the computing power of single processors. On the other hand, collections of processors have been arranged in multi-processors, multi-computers, and networks of geographically distributed machines.

After decades of continual improvement of single core chips trying to increase instruction-level parallelism, the majority of hardware manufacturers realized that the huge effort required for further improvements is no longer worth the benefits eventually achieved, notably because of power consumption. Thus, microprocessor vendors have shifted their attention to thread-level parallelism by designing chips with multiple internal cores, known as multi-cores (or chip multiprocessors). More generally, *parallelism* at multiple levels is now the driving force of computer design across all classes of computers, from small desktop workstations to large warehouse-scale computers.

We briefly recap Hennessy and Patterson's review of existing parallel computing platforms [84]. Following Flynn's taxonomy [80], we can define two main classes of architectures supporting parallel computing:

- Single Instruction Multiple Data (SIMD): the same instruction is executed by multiple processors on different data streams. SIMD computers support *data-level parallelism* by applying the same operations to multiple items of data in parallel;
- Multiple Instruction Multiple Data (MIMD): each processor fetches its own instructions and operates on its own data, and generally targets

task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable to larger classes of problems, but it is inherently more expensive than SIMD.

The MIMD class can be further subdivided into two subclasses:

- Tightly coupled MIMD architectures, which exploit thread-level parallelism since multiple cooperating threads operate in parallel on the same execution context (e.g., multi-cores, discussed in Sect. 2.1.2, and many-cores, discussed in Sect. 2.1.3);
- Loosely coupled MIMD architectures, which exploit parallelism at coarser grain, where many independent tasks can proceed in parallel with little need for communication or synchronization (e.g., clusters, discussed in Sect. 2.1.4).

Although the SIMD/MIMD is a common classification, it is becoming more and more coarse, as many processors are nowadays "hybrids" of the classes above. For instance, in recent years, one of the most popular approach specifically targeting data-level parallelism consists in the use of GPUs for general-purpose computing, known as the General-Purpose computing on Graphics Processing Units (GPGPU) paradigm. Most of the GPGPU processors (cf. Sect. 2.1.3) are based on Single Instruction Multiple Thread (SIMT), a model of parallelism similar to SIMD, in which the same instruction is *possibly* executed by multiple processors on different data streams.

We proceed by providing a brief survey of the parallel platforms that can be found nowadays in HPC environments, in ascending order of parallelism degree. in Sect. 2.1.1, we discuss SIMD computers, the earliest form of parallel platforms. In Sect. 2.1.2, we discuss Symmetric Multiprocessors (SMPs), the most common form of standalone parallel computer. In Sect. 2.1.3, we discuss many-core computers, that are usually attached to standalone computers for accelerating specific computations. Finally, in Sect. 2.1.4, we discuss distributed systems, consisting in networks of interconnected computers, thus possibly aggregating all the discussed forms of parallelism.

2.1.1 SIMD computers

The first use of SIMD instructions was in 1970s with *vector supercomputers* such as the CDC Star-100 and the Texas Instruments ASC. Vector-processing architectures are now considered separate from SIMD machines: vector machines processed vectors one word at a time exploiting pipelined processors (though still based on a single instruction), whereas modern SIMD machines process all elements of the vector simultaneously [117].

Simple examples of SIMD computers are Intel SSE (Streaming SIMD Extensions) [87] for the x86 architecture. Processors implementing SSE (usually with a dedicated unit) can perform simultaneous operations on multiple operands in a single register. For example, SSE instructions can simultaneously perform eight 16-bit operations on 128-bit registers.

SSE evolved into AVX (Advanced Vector Extensions). Specifically, AVX-512 are 512-bit extensions to the 256-bit AVX instructions for x86 Instruction

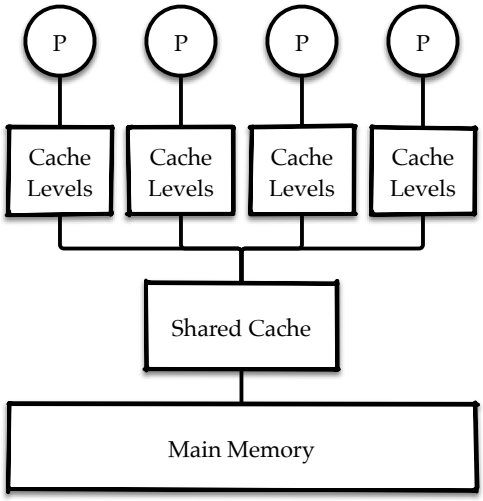


FIGURE 2.1: Structure of a 4-processor SMP.

Set Architecture (ISA), proposed by Intel in July 2013, and supported in Intel's Xeon Phi x200 (a.k.a. Knights Landing) processor [88]. Programs can pack various number of elements all within a single 512-bit vectors—e.g., eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers. This enables processing twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.

Advantages of such approaches consist in almost negligible overhead and low hardware cost. However, they are difficult to integrate into existing code, as they actually require writing code in assembly language. Moreover, although many compilers provide automatic vectorization (e.g., collapsing independent loop iterations in a single SIMD instruction), the applicability of this technique is limited to extremely regular code.

2.1.2 Shared-Memory Multiprocessors

Thread-level parallelism implies the existence of multiple program counters, hence it must be exploited primarily through MIMDs. Threads can also be used to support data-level parallelism, but some overhead is introduced at least by thread communication and synchronization. This overhead means the *grain size* (i.e., the ratio of computation to the amount of communication), a key factor for efficient exploitation of thread-level parallelism, must be properly selected.

The most common MIMD computers are multiprocessors, defined as computers consisting of *tightly coupled* processors that share memory. SMPs typically feature small numbers of cores (nowadays from 12 to 24), where processors can share a single *centralized* memory, to which they all have equal access (Fig. 2.1). Among them, single-chip systems with multiple cores are known as *multi-cores*. In multi-core chips, the memory is effectively centralized, and all existing multi-cores are SMPs. SMP architectures are also sometimes called Uniform Memory Access (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks.

The alternative "asymmetric" design approach consists in using multiprocessors with physically distributed memory. To support larger numbers of processors, memory must be distributed rather than centralized—otherwise, the memory system would not be able to support the bandwidth demands of processors without incurring excessively long access latency. Such architectures are known as Non-Uniform Memory Access (NUMA), since the access time depends on the location of data in memory.

Multiprocessors usually support the caching of both shared and private data, reducing the average access time as well as the required memory bandwidth. Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which could end up seeing two different values. This problem is generally referred to as the *cache coherence problem* and several *protocols*, referred to as cache coherence protocols, have been designed to guarantee cache coherence.

From the performance perspective, guaranteeing cache coherence introduces a number of pitfalls. For instance, *false sharing* is a subtle source of cache miss, which arises from the use of an invalidation-based coherence algorithm and multiple words block. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not have occurred if the block consisted of a single word.

On top of cache coherence, each ISA specifies a *memory consistency model*, that addresses the general problem of defining the behavior of multiple processors with respect to the memory they are attached to.

We discuss cache coherence and memory consistency in more detail in Sects. 2.3.1 and 2.3.2, respectively.

2.1.3 Many-Core Processors

Many-Core processors are specialized multiprocessors designed to exploit a high degree of parallelism, containing a large number of simpler, independent processor cores. They are often referred to as *hardware accelerators*. A many-core processor contains at least tens—if not hundreds—of cores and usually distributed memory, which are connected (but physically separated) by an interconnect that has a communication latency of multiple clock cycles [118]. A multiprocessor architecture (e.g., a multi-core) equipped with hardware accelerators is a form of *heterogeneous architecture*.

We can compare multi-core to many-core processors as follows:

- Multi-core: a SMP architecture containing tightly coupled identical cores that all share memory, where caches coherence is guaranteed.
- Many-Core: specialized multiprocessors designed for a high degree of parallel processing, containing a large number of simpler, independent processor cores (e.g., tens up to thousands) with a reduced cache coherency to increase performance. Indeed, as the core count increases, cache coherency is unlikely to be sustained by the hardware [48].

Although broadly different types of accelerators have been proposed, they do share some features. For instance, cores in many-core processors

are typically slower with respect to multi-core processors, and the high performance is obtained by high level of parallelism rather than high speed of each core. Moreover, data transfer from the host to the accelerator is typically slower than the memory bandwidth within the host processor, therefore exploiting locality is mandatory to achieve good performance.

In the following, we recap some of the accelerators that can be found in several HPC scenarios.

- GPUs include a large number of small processing cores (from hundreds to thousands) in an architecture optimized for highly parallel workloads, paired with dedicated high performance memory. They are accelerators, used from a general purpose CPU, that can deliver high performance for some classes of algorithms.

- Intel Xeon Phi is a brand name given to a series of massively-parallel multi-core co-processors designed and manufactured by Intel, targeted at HPC. A crucial component of each co-processor's core is the Vector Processing Unit (VPU), for the execution of AVX-512 SIMD instructions. Currently, the *Knights Landing* family (the successor of the first Xeon Phi) features 64 to 72 cores, with 4 thread contexts per core, organized as interconnected *tiles*, each containing two CPU cores and two VPUs per core (i.e., four VPUs per tile).

- Field Programmable Gate Arrays (FPGAs) are semiconductor devices based around a matrix of configurable logic blocks, connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. The FPGA configuration is generally specified using a hardware description language, similar to that used for an application-specific integrated circuit.

2.1.4 Distributed Systems, Clusters, and Clouds

In contrast with shared-memory architectures, *distributed systems* look like individual computers, each owning exclusive access to its private memory, connected by a network. With respect to Flynn's categorization, distributed systems are examples of loosely coupled MIMDs.

Clusters are generally defined as homogeneous distributed systems, in the sense that the computers in the network are identical. Therefore, both program binaries and data are represented in the same manner (e.g., endianness) on each computer in the cluster, thus allowing the computers to exchange and process data without introducing additional layers of data conversion. Moreover, users of a cluster typically have *direct* access to the computing resources, possibly mediated by some allocation mechanism. The mentioned aspects leads clusters to be considered the reference model of distributed system in the context of HPC.

Conversely, large-scale distributed systems are typically heterogeneous and they form the basis for *cloud computing*, in which an infrastructure is offered by providers according to a *pay-per-use* business model. In a common cloud model, referred to as Infrastructure-as-a-Service (IaaS), providers offer computers—either physical or (more often) virtual machines—and other resources on-demand. End users are not required to (or, depending on the

viewpoint, allowed to) take care of hardware, power consumption, reliability, robustness, security, and the problems related to the deployment of a physical computing infrastructure.

2.2 Parallel Programming Models

Shifting from sequential to parallel platforms, as discussed in the previous section, does not always translate into greater performance. For instance, sequential code (i.e., code not exploiting any form of parallelism) will get no performance benefits from a workstation equipped with a quad-core CPU: in such a case, running sequential code is wasting $\frac{3}{4}$ of the machine computational power. Developers are then faced with the challenge of achieving a trade-off between performance and human productivity (total cost and time to solution) in developing and porting applications to parallel platforms.

Therefore, effective parallel programming happens to be a key factor for exploiting parallel computing, but efficiency is not the only issue faced by parallel programmers: writing parallel code that is portable on different platforms and maintainable are also issues that parallel programming models should address.

We proceed by recalling, in Sect. 2.2.1, the forms of parallelism that can be expressed in the most common parallel programming models and, in Sect. 2.2.2, some shared memory and communication models, regarded as orthogonal aspects in this thesis. Finally, in Sects. 2.2.3 and 2.2.4, we provide a survey of low-level and high-level parallel programming models, respectively, with a focus on the C++ realm.

2.2.1 Types of Parallelism

Parallel programming models allow to express parallelism in programs. In the following, we recall some common types of parallelism, as provided by the models proposed in various well-established parallel programming models.

- *Task Parallelism* consists of running independent computations (i.e., tasks) on different executors (cores, processors, etc.), according to a task-dependency graph. Tasks are concretely processed by threads or processes, which may communicate with one another as they execute. Communication takes place usually to pass data from one thread to the next as part of a graph.

- *Data Parallelism* is a method for parallelizing a single task by processing independent data elements in parallel. The flexibility of the technique relies upon stateless processing routines, implying that the data elements must be fully independent. Data Parallelism is often realized in terms of *Loop-level Parallelism*, where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control (according to the model *co-begin/co-end*) and concurrently executed.

- *Stream Parallelism* is a method for parallelizing the execution (aka. filtering) of a stream of tasks by segmenting each task into a series of

sequential[1] or *parallel* stages. This method can be also applied when there exists a *total* or *partial* order in a computation, preventing the use of data or task parallelism. This might also come from the successive availability of input data along time (e.g., data flowing from a device). By processing data elements in order, local state may be either maintained in each stage or distributed (replicated, scattered, etc.) along streams. Parallelism is achieved by running each stage simultaneously on *subsequent* or *independent* data elements.

2.2.2 Memory and Communication Model

From a programming perspective, memory is represented in terms of *address spaces*, accessed by processing units (e.g., processes or threads). In the space of parallel programming models, *shared-memory* models yield programs composed of a single address space, shared by all processing units. This shared space is also referred as Global Address Space (GAS). Conversely, *distributed-memory* models yield programs in which each executor is attached to a private address space, that cannot be accessed by any other executor.

A further categorization of parallel programming models can be formulated in terms of the *communication* model among processing units. In models based on *message passing*, communications among processing units are performed via explicit messages. When processing units need to exchange data among each other, this exchange is done by *sending* and *receiving* messages, which typically requires cooperative operations among the two units involved in the communication (namely, the *sender* and the *receiver*). This aspect induces tight coupling of processing units: any send operation must have its corresponding receive operation, otherwise a deadlock can occur, since the process can be waiting indefinitely for completion.

In distributed-memory models, message passing is the only viable option for exchanging data among processing units, therefore all distributed-memory models are based on message passing as communication model. Note that exchanging data in the form of messages induces some extra effort. For instance, it typically requires data to be *serialized* and de-serialized, which introduces complexity and possible performance penalties, in particular when working with complex data structures.

In shared-memory programming, processing units share the GAS, in which they communicate implicitly via load/store primitives. The memory locations of the GAS can be used to effectively exchange data among threads, but the access to memory locations must always be coordinated (e.g., by locks/semaphores) to prevent data races, starvations or deadlocks. Some shared-memory programming model also provides communication by means of message passing, that can be exploited as an alternative synchronization and coordination mechanism, with respect to semaphores and atomic instructions.

We remark that *shared-memory* models can be implemented on top of distributed platforms (cf. Sect. 2.1.4), with the assistance of software protocols. In the simplest realization of this approach, which is historically referred as DSM, the memory interface (i.e., the Application Programming

[1] In the case of total sequential stages, the method is also known as *Pipeline Parallelism*.

Interface (API) provided to processing units to access the GAS) is the same as a shared-memory model based on physically shared memory. According to DSM models, the GAS can be accessed by the processing units using a plain load/store API and some form of strong consistency (cf. 2.3.2) is guaranteed. Although retaining the simplicity of shared-memory models when programming distributed platforms seems attractive, the DSM approach faded away, mainly because of the inherent limits to scalability imposed by keeping memory consistent. Nevertheless, in recent years, a number of GAS models have been proposed for distributed platforms, both enriching the memory interface and relaxing the consistency model. For instance, the PGAS paradigm (cf. Sect. 2.2.3) revamped the DSM approach by adding syntactic mechanisms to control data locality.

2.2.3 Low-Level Programming Models

We denote as *low-level* parallel programming models that provide the programmers with a thin abstraction over the underlying parallel platform to be programmed. For instance, on top of a shared-memory multiprocessor, a low-level programming model typically provides primitives for managing the lifetime of processing units (i.e., threads), their synchronization and data sharing, typically accomplished through critical regions accessed in mutual exclusion. Low-level languages are usually extensions to well-established sequential languages, such as C/C++, Java, or Fortran, by means of external libraries, linked at compile time to the application source code (e.g., *Pthreads*, *MPI*), or enriched with specific constructs (e.g. C++ Threads).

Shared-Memory Platforms

POSIX Threads (or Pthreads) [43], one of the most used low-level parallel programming API for shared-memory environments, are defined by the POSIX.1c standard, Threads extensions (IEEE Std 1003.1c-1995). They are present in every Unix-like operating system (Linux, Solaris, Mac OS X, etc.) and other POSIX systems, giving access to OS-level primitives for creation and synchronization of threads.

From the programming model perspective, with respect to the categorization discussed above, Pthreads provides shared-memory programming. Since no primitives are provided for explicit communication among threads, the user has to implement implicit communication by means of *concurrent data structures* (i.e., data structures accessed concurrently by multiple threads) as, for instance, First-In First-Out (FIFO) queues.

Since Pthreads is a C library, it can be used in C++ programs as well. However, a well-known report by Boehm [37] provides specific arguments that a pure library approach, in which the compiler is designed independently of threading issues, cannot guarantee correctness of the resulting code, for instance with respect to the parallel memory model (cf. Sect. 2.3). The report shows simple cases (e.g., concurrent modification, adjacent data rewriting and register promotion) in which a pure library-based approach is incapable of expressing a correct and efficient parallel algorithm. For these and similar reasons, the C++11 standard, published in 2011, introduced multithreaded programming. We remark that ISO C++ is completely

independent from POSIX and it is provided also in non-POSIX platforms. In a similar fashion, *Java* provides multi-threading for writing parallel applications according to a shared-memory programming model [113].

In contrast to Pthreads, multithreading in ISO C++ includes a complete *parallel memory model* (abbr. memory model), that defines the behavior of parallel programs with respect to memory accesses. The C++ memory model has been formulated by Boehm et al. [38], on the same line as the Java memory model [99]. As we discuss in more detail in Sect. 2.3.2, the key concept is guaranteeing safe memory behaviors for safe programs, where program safeness is ensured by either avoiding concurrent stores to the same memory location or mediating them by special operations.

CUDA (Compute Unified Device Architecture) [111] is the reference language for programming Nvidia GPUs, according to the GPGPU paradigm. Although there have been recent efforts (e.g., Nvidia Thrust library [112], discussed in Sect. 6.1.2) for partially reducing the gap between high computational power, provided by GPUs, and easiness of programming, we still regard CUDA as a low-level parallel programming approach, since, in the general case, the user has to deal with close-to-metal aspects like memory allocation and data movement between the GPU and the host platform. Moreover, CUDA programming is thread-centric, which induces non-trivial consequences when programming data-parallel applications, as discussed by Drocco *et al.* [74].

OpenCL (Open Computing Language) [93] is an API designed to write parallel programs that execute across heterogeneous architectures, including GPUs. It is implemented by different hardware vendors such as Intel, AMD, and Nvidia, thus making programs portable with respect to hardware accelerators. For instance, OpenCL applications are seamlessly reverted to the CPU for execution when there is no GPU in the system, and its portability makes it suitable for hybrid CPU/GPU environments. Moreover, OpenCL allows the implementation of applications onto FPGAs, allowing software programmers to write hardware-accelerated kernel functions in OpenCL C, an ANSI C-based language with additional OpenCL constructs. OpenCL represents an extension to C/C++ but, as CUDA, must be considered a low-level language, focusing on low-level features management rather than high-level parallelism exploitation patterns. Nevertheless, as for CUDA, recent efforts (e.g., SYCL [94]) aim at raising the programming level by hiding low-level details.

Distributed Platforms

MPI [115] is a language-independent communication protocol, as well as a message-passing API, that supports point-to-point and collective communication. Many general-purpose programming languages have bindings to MPI functionalities, among which: C, C++ (e.g., notably, Boost.MPI [40]), Fortran, Java, and Python. Mainly targeted to distributed architectures, MPI offers specific implementations for almost any high-performance interconnection network. At the same time, implementations exist that allow to use MPI even on standalone multiprocessor systems.

From the programming model perspective, with respect to the categorization discussed above, MPI provides message-passing communication on top of a distributed-memory model. Moreover, programming in MPI

implicitly follows the Single Program Multiple Data (SPMD) paradigm [63], in which all processing units execute the same program, each operating on its local chunk of data.

MPI allows to manage synchronization and communication functionalities among a set of processes, and provides mechanisms to deploy a virtual topology on top of the system upon which the program is executing. These features, supported by a rich set of capabilities and functions, clearly require high programming and networking skills. Nevertheless, MPI has long been the lingua franca of HPC, supporting most of the supercomputing scientists and engineers have relied upon for the past two decades.

Shifting from the distributed-memory to the shared-memory programming model, UPC (Unified Parallel C) [77] is a long-standing example of the PGAS approach, expressed through a C language extension. In UPC, 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 to a partition of the 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. Since PGAS is a shared-memory model, threads access shared memory addresses concurrently through standard read and write instructions. This programming model is still a low-level shared-memory environment, and uses barriers and locks to synchronize the execution flow.

Several PGAS languages and libraries have been proposed. Such languages can be categorized along a number of orthogonal dimensions, including relationship between local and shared data, model of communication among processors, representation of global memory reference (e.g., global pointers), supported forms of parallelism, and level of memory consistency. Among the most successful PGAS languages, we can mention *Global Arrays* [110], *UPC (Unified Parallel C)* [77], *UPC++* [131], *Chapel* [45], and *X10* [46].

On the same line as the DSM paradigm, several Global Object Space (GOS) languages have been proposed. For instance, *Charm++* [91] is a C++ variant providing objects with parallel semantics and supporting compilation for both shared-memory and distributed platforms. Similarly, *ADHOC (Adaptive Distributed Herd of Object Caches)* [15] provides virtualization of local memories (i.e., partitions) into an unique distributed object repository. Also the popular memcached [102] caching system is based on distributed memory objects, thus it can be regarded as a realization of the GOS paradigm.

2.2.4 High-level Programming Models

Parallel programming is intimately related to HPC environments, where programmers write low-level parallel code to retain complete control over the underlying platform, allowing them to manually optimize the code in order to exploit at best the parallel architecture. This programming methodology has become unsuitable with the fast move to heterogeneous architectures, that encompass hardware accelerators, distributed shared-memory systems and cloud infrastructures, highlighting the need for proper tools to easily implement parallel applications. Indeed, it is widely acknowledged that the main problem to be addressed by a parallel programming model

is *portability*: the ability to not only compile and execute the same code on different architectures [65], but also—and generally even more complex—the challenge of performance portability, that is, implementing applications that scale on different architectures.

A *high-level* approach to parallel programming is a better way to go to address this problem, so that programmers can build parallel applications and be sure that they will perform reasonably well on the wide range of parallel architectures available today [120]. For instance, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimize cache flushing or maximize load balancing of cores. Synchronization primitives can be also abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc. Intel *TBB (Threading Building Blocks)* [89], *OpenMP (Open Multi-Processing)* [116], and *Cilk* [50] all provide those kinds of abstraction, each in its own way. A complete review of these parallel programming models is proposed by Sanchez *et al.* [119], where it is provided a comparative study and evaluation of OpenMP, TBB, Cilk, Intel *ArBB (Array Building Blocks)* and OpenCL. The study covers several capacities, such as task deployment, scheduling techniques, or programming language abstractions.

Arguably, the most basic form of abstraction to raise the level of abstraction and reduce the programming effort is based on *tasks*. As we discuss in more detail in Sect. 6.2.1, any parallel computation can be described in terms a graph of tasks, where independent tasks (i.e., not linked) can be performed in parallel. Following this principle, a number of frameworks for task-based parallel programming have been proposed. For instance, OCR (Open Community Runtime) [101, 71, 29] is a recently proposed task-based runtime system, targeting future extreme-scale applications.

Further along the direction of raising the level of abstraction, notable results have been achieved by the *algorithmic skeleton* approach [52] (aka. *pattern-based* parallel programming), that has gained popularity after being revamped by several successful parallel programming frameworks. Algorithmic skeletons have been initially proposed by Cole [53] to provide predefined parallel computation and communication patterns, hiding parallelism management from the user. *Algorithmic skeletons* capture common parallel programming paradigms (e.g., Map+Reduce, ForAll, Divide and Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [12]. Ideally, algorithmic skeletons address the difficulties of parallel programming (i.e., concurrency exploitation, orchestration, mapping, tuning) by moving them from the application design to development tools, which is done by capturing and abstracting common paradigms of parallel programming and providing them with efficient implementations. This idea can be considered at the core of *structured parallel programming*: expressing the parallel code as a composition of simple "building blocks".

Over the last two decades, many skeletons have been proposed, covering many different usage schema of the three classes of parallelism, on top of both message passing [54, 64, 18, 11] and shared memory [5, 89] models. In the following, we briefly present some frameworks for high-level parallel

programming, from the C/C++ world, focusing in particular on skeleton-based approaches. For a broader presentation of algorithmic skeletons, see the survey by González-Vélez and Leyton [83].

Skeletons for Shared-Memory Platforms

OpenMP [116] is widely considered the *de facto* standard API for shared-memory parallel programming. OpenMP is an extension that can be supported by C, C++ and Fortran compilers and that defines an "accelerator-style" programming, where the main program is run sequentially while code is accelerated at specific points, in "parallel regions", specified using special preprocessor instructions known as `pragma`s. Compilers that do not support specific `pragma`s can ignore them, making an OpenMP program compilable and runnable on every system with a generic sequential compiler.

While Pthreads are low-level and require the programmer to specify every detail of the behavior of each thread, OpenMP allows to simply state which block of code should be executed in parallel, leaving to the compiler and run-time system the responsibility to determine the details of the thread behavior. In addition to the `parallel_for` construct, that can be used to express data parallelism, OpenMP has been recently enriched with `pragma`s targeting task parallelism, allowing the user to simply identify which block of code should be considered a separate task, leaving to the runtime the burden of efficient scheduling and execution of tasks.

TBB [89] defines a set of high-level parallel patterns that permit to exploit parallelism independently from the underlying platform details and threading mechanisms. It targets shared-memory multi-core architectures, and exposes parallel patterns for exploiting data parallelism, stream parallelism, as well as task parallelism. For example, the `parallel_for` and `parallel_for_each` functions may be used to parallelize independent iterations of a definite (for) loop. C++11 *lambda* expression can be used as arguments to these calls, so that the loop body function can be described as part of the call, rather than being separately declared. The `parallel_for` uses a divide-and-conquer approach, where a range of iterations is recursively split into sub-ranges until each sub-range is sufficient small that it can be processed as a separate task using a serial for loop.

SkePU [78] is an open-source framework for skeleton programming on multi-core CPUs and multi-GPU systems. It is a C++ template library with data-parallel and task-parallel skeletons (map, reduce, map-reduce, farm) that also provides generic container types and support for execution on multi-GPU systems, both with CUDA and OpenCL.

SkelCL [122] is a skeleton library targeting OpenCL. It allows the declaration of skeleton-based applications hiding all the low-level details of OpenCL. The set of skeletons is currently limited to data-parallel patterns—map, zip, reduce, and scan—and it is unclear whether skeleton nesting is allowed. A key limitation stems from the library's target, which is restricted to the OpenCL language: it likely benefits from the possibility to run OpenCL code both on multi-core and on many-core architectures, but the window for tunings and optimizations is restricted.

Similarly to SkePU and SkelCL, a set of parallel patterns in the form of a pattern-based library for OpenCL has been proposed in [70], where authors

exploring issues and opportunities encountered by attempts to provide patterns such as like parallel for-loops or pipelines. Due to very different performance characteristics of the different OpenCL devices, authors state that it is necessary to include some kind of dynamic work allocation technique, or adaptive static strategies.

GrPPI (Generic Parallel Pattern Interface) [69] is a generic high-level pattern interface for stream-based C++ applications. Thanks to its high-level C++ API, this interface allows users to easily expose parallelism in sequential applications using already existing parallel frameworks, such as C++ threads, OpenMP, and TBB. It is implemented using C++ template metaprogramming techniques to provide interfaces of a generic, reusable set of parallel patterns without incurring runtime overheads. GrPPI targets the following stream parallel processing patterns: *Pipeline*, *Farm*, *Filter*, *Stream-Reduce*, and *Stream-Iteration*. Parallel versions of the proposed interfaces are implemented by leveraging C++11 threads and OpenMP, as well as the pattern-based parallel framework Intel TBB. As for the parameter functions, they are specified as user lambdas.

FastFlow [17] is a parallel programming framework originally designed to support streaming applications on cache-coherent multi-core platforms. Since it plays a key role in our project, it is described more in detail in Section 2.4.2.

Skeletons for Distributed Platforms

HPF (High Performance Fortran) [92] is among the first attempts of raising the abstraction level in the context of distributed-memory programming. It provides an annotation-based syntax similar to OpenMP focused on data parallelism, thus allowing to express parallel iterative computations in a compact way.

P^3L [59] is one of the earliest proposals for pattern-based parallel programming. P^3L is a skeleton-based coordination language that manages the parallel or sequential execution of C code. It comes with a proper compiler for the language, and uses implementation templates to compile the code into a target architecture. P^3L provides patterns for both stream parallelism and data parallelism.

SKELib [60] builds upon the contributions of P^3L by inheriting, among other features, the template system. It differs from P^3L because a coordination language is no longer used, and skeletons are provided as a C library. It only offers stream-based skeletons (namely farm and pipe patterns).

SkeTo [100] is a C++ library based on MPI that provides skeletons for distributed data structures, such as arrays, matrices, and trees. The current version is based on C++ expression templates, used to represent part of an expression where the template represents the operation and parameters represent the operands to which the operation applies.

Muesli [49] is a C++ template library that supports SMPs and distributed architectures using MPI and OpenMP as underlying parallel engines. It provides data parallel patterns such as map, fold (i.e., reduce), scan (i.e., prefix sum), and distributed data structures such as distributed arrays, matrices, and sparse matrices. Skeleton functions are passed to distributed objects as pointers, since each distributed object has skeleton functions as

internal member of the class itself. The programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons, if available.

Other High-Level Programming Frameworks

In addition to skeleton-based approaches, a number of frameworks, targeting both shared-memory and distributed platforms, have been proposed as DSLs. For instance, *Google MapReduce* [68] and *Thrill* [36] have been proposed within the domain of data analytics in C++. From the perspective of categorizing parallel programming models, we regard such approaches as high-level models, in which even the parallelism itself is hidden by the APIs, usually yielding programs that look like plain sequential code.

Moreover, in recent years, a number of frameworks have been proposed that provide a task-based Run-Time System (RTS), coupled with a higher-level programming model, to hide the complexity of managing task graphs. This class of frameworks includes OCR [101], OmpSs [42], CAF (C++ Actor Framework) [47], HPX (High Performance ParalleX) [90], UPC++[2], and Legion [32]. From the API perspective, for instance, both HPX and UPC++ focus on tight integration with task-based constructs in modern C++ (e.g., `async`), whereas the CAF API is based on the actor model [2].

2.3 Parallel Memory Models

A *memory model*—also known as a *memory consistency model*—defines the semantics of a shared-memory system. That is, the memory model specifies the values that a shared variable read in a multi-threaded program is allowed to return. The memory model affects programmability, performance, and portability by constraining the transformations that any part of the system may perform. In short, as expressed by Sorin et al. [121], such "models define correctness so that programmers know what to expect and implementors know what to provide".

A memory consistency model typically defines the shared memory behavior in terms of loads and stores (memory reads and writes), without any reference to caches. However, cache coherence can play a key role in implementing the most basic form of consistencies. Therefore, we first describe cache coherence, in Sect. 2.3.1, before discussing memory consistency, in Sect. 2.3.2.

2.3.1 Cache Coherence

A cache coherence problem can arise if multiple actors (e.g., cores) have access to a copy of a datum in their cache and at least one actor performs a write on that datum: the value written by the writing core must be returned, when a read is performed, to the other cores. Access to a wrong (incoherent) value of the datum can be precluded using a *coherence protocol*, which must ensure certain appropriate conditions.

More precisely, Sorin et al. [121] characterize coherence using the following two conditions:

[2]We already mentioned UPC++, in Sect. 2.2.3, as a low-level PGAS framework; indeed, as many other PGAS frameworks we mentioned, UPC++ can be regarded from both low-level (i.e., DSM-oriented) and high-level (i.e., C++ task-oriented) perspectives.

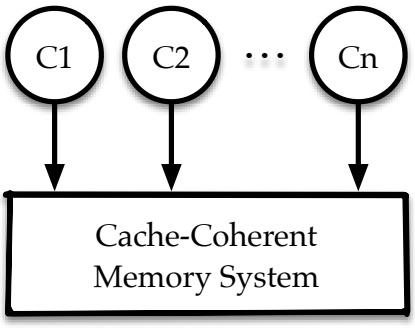


FIGURE 2.2: Implementing SC with cache coherence; cores issue memory instructions in parallel, according to the respective program order (from Sorin *et al.* [121]).

- The Single Writer Multiple Reader (SWMR) invariant: given a memory location, at any moment in time 1) there is a single core that may *write* (and also read) it or 2) there are some number of cores that may *read* it. As a consequence, there is never a time in which that memory location may be written by a core and read or written by other cores. Thus, the lifetime of this memory location is divided into *epochs*. Within each epoch, it happens that: 1) a single core has read and write access or 2) zero, one or more cores have read-only access.
- The Data-Value (DV) invariant: after a write by a core, the value is correctly propagated from an epoch to the next, that is, the value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

2.3.2 Memory Consistency

Sorin et al. define a memory consistency model as follows [121]:

> A *memory consistency model*, or, more simply, a *memory model*, is a specification of the allowed behavior of multi-threaded programs executing with a shared memory. For a multi-threaded program executing with specific input data, it specifies what values dynamic loads may return and what the final state of memory is.

The most basic form of consistency is Sequential Consistency (SC), in which "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" [95]. Hence, the execution order *within* a single processor (or thread) is the same as the program order, while the execution order of program *between* processors (or threads) is undefined.

Although they may appear similar, there is a key difference between coherence and consistency: whereas coherence is specified *on a per-memory location*, consistency is specified *with respect to all memory locations*—that

is, also in terms of possible interactions among distinct locations. Furthermore, as discussed by Sorin et al. [121] and as illustrated in Fig. 2.2, cache coherence can be used to provide completely parallel implementations of SC.

Since the introduction of explicit constructs for expressing parallelism (e.g., threads), memory consistency has become a first-class citizen of programming languages. Indeed, memory consistency models are part of the semantics, with respect to parallelism, of such parallelism-aware programming languages. For instance, SC is guaranteed for a well-formed subset of C++ and Java multi-threaded programs through their respective memory models [38, 99]. In particular, SC is guaranteed for race-free programs, where a race is defined as a situation in which the same memory location may be read and written concurrently by different threads.

For performance gains, to fully utilize resources, modern CPUs often execute instructions out of order. Furthermore, the compiler can also optimize the code by reordering instructions. Since the hardware enforces instructions integrity, this cannot be noticed within a single thread execution. However, in a multi-threaded execution, reordering may lead to unpredictable behaviors. SC also restricts many common compiler and hardware optimizations and to overcome the performance limitations of this model, hardware vendors and researchers have proposed several relaxed memory models, as reported in [1], up to the extreme totally *relaxed* model, in which any memory ordering has to be forced by means of special *fence* instructions. Between SC and the totally relaxed model, a plethora of partially relaxed models have been proposed at ISA level, including, for instance, the widespread Total Store Order (TSO), in which write-to-read dependency is dropped to allow store buffers. We remark that the mentioned C++ memory model encompasses different ISA memory models, by optionally associating a consistency level to `atomic` memory accesses.

2.4 Libraries Used by our Implementation

In this section, we provide some details about the libraries that we use for implementing the stack in Fig. 1.1.

In Sect. 2.4.1, we briefly present smart pointers, an approach to automatize dynamic memory management that we adopted in our implementation of smart global pointers (cf. Ch. 4). In Sect. 2.4.2, we describe the FastFlow library for structured parallel programming. Although we do not specifically use FastFlow in any layer of the stack, the parallel programming model that we propose (cf. Ch. 5) is based on the same ideas as FastFlow. Finally, in Sect. 2.4.3, we briefly describe the libfabric library for large-scale network programming, that we used at the very bottom of the stack to support arbitrary networking environments.

2.4.1 C++ Smart Pointers

Since most programming languages support dynamic memory allocation, the problem of automating memory allocation (and deallocation) has been around for decades. A commonly used approach, adopted for instance in Java, is referred as *garbage collection* and relies on a component (i.e., the

garbage collector) of the language RTS that periodically checks if some non-referenced memory exists and eventually reclaims it. Although drastically simplifying the programmer's task, who can indeed safely forget about freeing the allocated memory, garbage collection induces non-negligible performance costs. In particular, in case of multithreaded dynamic memory allocation, garbage collection usually implies some locking mechanism to coordinate the concurrency over the involved data structures, therefore exacerbating performance drawbacks. As a matter of fact, it is a common practice to switch off garbage collection whenever performance matters (e.g., in HPC environments), thus possibly exposing execution to the problem of memory explosion.

Smart pointers represent a dual approach with respect to garbage collection: instead of allowing non-referenced memory and relying on an additional software component to collect it, smart pointers *prevent* memory to become non-referenced. This invariant—at any time during the program execution, no non-referenced memory exist—is maintained by binding the lifetime of memory locations to the lifetime of their respective references. For instance, in an object-oriented context, such "active" references are objects (i.e., smart pointers) that implement a cooperative *reference counting* over the respective memory location they control. The counter for a memory location m is decremented when the *destructor* a smart pointer referencing m is called, thus realizing the mentioned location-reference binding.

C++ supports this approach, based on reference counting, by means of *shared* pointers—i.e., objects of the `shared_ptr` class. In addition to shared pointers, *unique* pointers—i.e., objects of the `unique_ptr` class—guarantee an additional *exclusiveness* invariant: at any time during the program execution, no two unique pointers exist that reference the same memory location. From the programming perspective, C++ smart pointers are implemented using templates and operator overloading, which generally allows those smart pointers to be used almost as ordinary pointers.

Based on this principle, it can be shown that the two following conditions hold during the lifetime of a program based on smart pointers:

- *Leak-freeness*: if a memory location m is allocated, then there exists a reference to m;
- *Dangling-freeness*: if a reference exists to a memory location m, then m is allocated.[3]

Considering a memory location m, the absence of leaks is guaranteed by freeing m before the last standing reference to m gets destructed, while the absence of dangling pointers is guaranteed implicitly by the existence of a reference to m.

2.4.2 FastFlow

FastFlow [62] is an open source programming framework for structured parallel programming, targeting shared-memory multi-core and supporting the exploitation of GPU accelerators. Its efficiency stems from the optimized implementation of the base communication mechanisms and from

[3]For the sake of simplicity, we consider the basic formulation of dangling-freeness, whereas more refined definitions (that still hold for C++ smart pointers) take into account, for instance, the *type* of the referenced value.

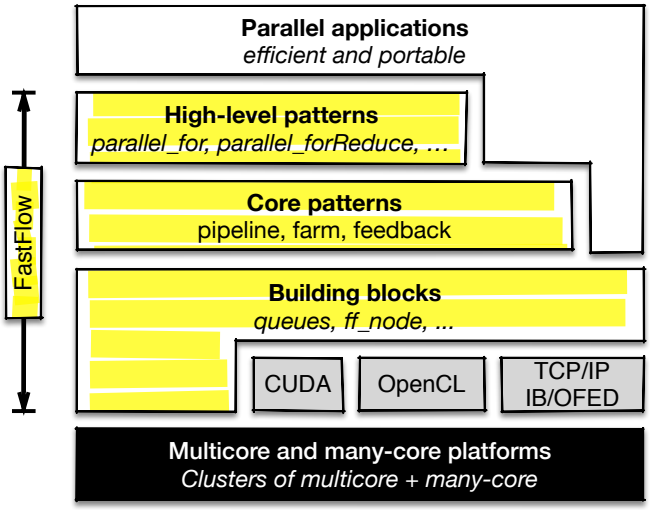


FIGURE 2.3: Layered FastFlow design.

its layered design (cf. Fig. 2.3), based on C++ templates. FastFlow provides a set of algorithmic skeletons addressing both stream parallelism (e.g., *farm* and *pipeline*) and data parallelism (e.g. map, stencil, reduce), along with their arbitrary nesting and composition [25]. Map, reduce, and stencil patterns can be run on multi-cores or can be offloaded onto GPUs. In the latter case, the user code can include GPU-specific code (i.e., CUDA or OpenCL kernels).

For instance, leveraging the farm skeleton, FastFlow exposes a *ParallelFor* pattern [61], where chunks of a loop iterations are streamed to be executed by the farm workers. Just like TBB, FastFlow's `parallel_for` pattern uses C++11 *lambda* expression as a concise way to create function objects: lambdas can "capture" the state of non-local variables, by value or by reference, and allow functions to be syntactically defined where and when needed.

From the performance viewpoint, one distinguishing feature at the core of FastFlow is that it supports lock-free (fence-free) Multiple Producer Multiple Consumer (MPMC) queues [20], thus providing low overhead high bandwidth multi-party communications on multi-core architectures for any *streaming network*, including cyclic graphs of threads. The key intuition underlying FastFlow is to provide the programmer with fast lock-free Multiple Producer Single Consumer (MPSC) queues and Single Producer Multiple Consumer (SPMC) queues—that can be used in pipeline to build MPMC queues—to support fast streaming networks.

Traditionally, MPMC queues are built as passive entities: threads concurrently synchronize (according to some protocol) to access data; these synchronizations are usually supported by one or more atomic operations (e.g., Compare-And-Swap) that behave as memory fences. FastFlow design follows a different approach: to avoid any memory fence, the synchronizations among queue readers or writers are arbitrated by an active entity (e.g., a thread). We call these entities *Emitter* (E) or *Collector* (C) according to their role; they actually read an item from one or more lock-free Single Producer Single Consumer (SPSC), queues and write onto one or more lock-free SPSC

queues. This requires a memory (pointer) copy but no atomic operations.

The advantage of this solution, in terms of performance, comes from the higher speed of the copy operation compared with the memory fence; this advantage is further increased by avoiding cache invalidation triggered by fences. This behavior also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, ensuring that the data is not concurrently written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues—for example, this is guaranteed in a stateless farm as well as many other cases.

Shared-memory FastFlow

The FastFlow implementation for shared-memory platforms provides two basic abstractions:

- *Process-component*, i.e., an active control flow entity, implemented by means POSIX threads;[4]
- *1-1 channel*, i.e., a communication channel between two components, realized with wait-free SPSC queues [16].

The 1-1 channel is "state of the art" in its class, in terms of both latency and bandwidth. For instance, the SPSC queue exhibits a latency down to 10 nanoseconds per message on a standard Intel Xeon @2.0GHz [16]. Dolz *et al.* [72] tested the correctness of FastFlow SPSC queue benign data races over a set of μ-benchmarks and real applications on a dual-socket Intel Xeon CPU E5-2695 platform.

FastFlow design is a layered one (see Fig. 2.3). On top of the mentioned basic abstractions, the bottom layer (*Building blocks* in Fig. 2.3) provides the following entities:

- *FastFlow node*, i.e., the basic unit of parallelism that is typically identified with a node in a streaming network. Such a node is used to encapsulate sequential portions of code implementing functions (i.e., process-components), as well as higher-level parallel patterns, such as pipelines and farms; From the API viewpoint, a FastFlow node is an object of the `ff_node` class;
- *Collective channel*, i.e., a communication channel among two or more `ff_node`s, of arbitrary type (e.g., SPSC, MPMC).

The second layer (*Core patterns* in Fig. 2.3) provides basic streaming pattern (i.e., farm and pipeline) and some common variants (e.g., ordering farm).

On top of core patterns, *High-level patterns* are provided to target different types of parallelism. For instance, `parallel_for` and `map` allow to express data parallelism in a similar manner as other popular frameworks, such as OpenMP and TBB.

[4] Porting to C++ threads is under investigation.

Pattern	Description
`unicast`	Send the input data to the (unique) connected peer (unidirectional point-to-point communication)
`broadcast`	Sends the input data to all connected peers
`scatter`	Sends different parts of the input data, typically partitions, to all connected peers
`onDemand`	Sends the input data to one of the connected peers, chosen at runtime on the basis of the actual workload
`fromAll`	(aka. *all-gather*) Receives different parts of the data from all connected peers combining them in a single data item
`fromAny`	Receives one data item from one of the connected peers

TABLE 2.1: Communication patterns among `ff_dnode`s.

Distributed FastFlow

An experimental extension, targeting distributed systems, has been implemented on top the ZeroMQ library [130]. Briefly, ZeroMQ is an LGPL open-source communication library providing the user with a socket layer that carries whole messages across various transports: inter-thread communications, inter-process communications, TCP/IP, and multicast sockets. ZeroMQ offers an asynchronous communication model, providing a quick construction of complex asynchronous message-passing networks with reasonable performance.

A `ff_dnode` (distributed `ff_node`) provides an external channel that can support various patterns of communication. The set of communication patterns allows one to provide exchange of messages among a set of distributed nodes, using well-known predefined patterns. The semantics of each communication pattern currently implemented are summarized in Table 2.1. Graphs of `ff_node`s can be connected by way of `ff_dnode`s, thus providing a homogeneous abstraction for programming both multi-core and distributed platforms.

2.4.3 Libfabric

OFI (OpenFabrics Interfaces) is a framework focused on exporting fabric[5] communication services to applications. Libfabric [97] is a core component of OFI, that defines the user API, enabling a tight semantic link between applications and underlying fabric services. More specifically, libfabric software interfaces have been co-designed with hardware providers (the bottom layer of the stack in Fig. 2.4) with the goal of giving access to different hardware for HPC users and applications.

[5] *Fabric* is an industry term to denote a network of interconnected devices in a tightly coupled environment.

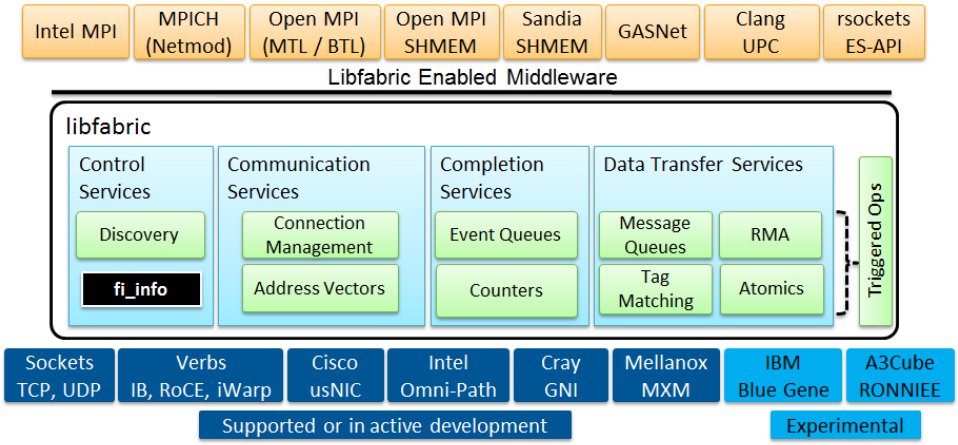


FIGURE 2.4: OFI interfaces overview [97].

A distinguishing feature of libfabric is that it is agnostic with respect to the underlying hardware provider, thus allowing programmers to write applications that can exploit any supported hardware. Based on this principle, we use the libfabric API for implementing the communication among GAM executors. In this setting, considering Fig. 2.4, the GAM runtime sits among the "libfabric-enabled middlewares", at the same level as MPI or UPC.

Libfabric provides two different APIs for transferring data among network nodes (the "Data Transfer Services" block on the right of Fig. 2.4): Message Queues and RMA. According to the former API, usually referred as *two-sided* communication, nodes communicate via intermediate queues by means of send and receive primitives, as in any message-passing environment. With the latter API, usually referred as *one-sided* communication, nodes exchange data by accessing memory locations from some shared space. For both APIs, libfabric enforces *asynchronism* by means of user-level notifications (the "Completion Services" block in the middle of Fig. 2.4), through which the user can query the runtime about the completion of issued data transfers, for instance, to safely reuse memory involved in transfers.

In addition to asynchronous operations, libfabric focuses its support on HPC environments through a number of design choices, described in detail in the "High Performance Network Programming with OFI" guide [114]. Among these, we based the implementation of GAM topologies on connection-less communication ("Address Vectors" within the "Communication Services" block in Fig. 2.4), that targets large-scale environments by reducing the amount of memory required to maintain large address look-up tables, thus eliminating expensive address resolution.

Summary

In this chapter, we provided a review of the most common parallel computing platforms and programming models for such platforms, with a focus on HPC environments. We also provided a brief review of parallel memory models, in particular SC, that we used to characterize the memory model

proposed in this thesis. Finally, we described the libraries that we exploited in developing the contributions of this thesis, namely C++ smart pointers, the FastFlow framework for structured parallel programming, and the libfabric library for large-scale, high-performance networking.

Chapter 3

Global Asynchronous Memory

In this chapter, we present the first novel contribution in this thesis.

We introduce the GAM programming model, based on a memory space shared among a set of executors (i.e., a GAS). A GAM memory location is either *public* or *private*. Public memory is accessed in a *single-assignment* fashion, whereas private memory is accessed *exclusively* by the respective *owner*. Therefore, GAM programs are Data Race Free (DRF) by construction. By proposing GAM, we advocate to trade off some expressiveness—GAM memory is more limited than an arbitrary load/store memory—in exchange of an efficient yet user-friendly memory consistency model (i.e., SC).

With respect to the categorization in Sect. 2.2, GAM is a shared-memory model, thus based on a shared address space. Moreover, GAM provides message-passing communication *along with* shared-memory primitives, by which executors exchange *capabilities* over memory locations, thus overcoming the traditional dichotomy between shared-memory and message-passing paradigms.

We materialize the proposed GAM model in a C++ library, implemented on top of libfabric (cf. Sect. 2.4.3) to target multiple networking hardware in the context of large-scale HPC environments.

This chapter proceeds as follows. In Sect. 3.1, we introduce GAM as abstract model, together with an operational semantics for GAM programs, in Sect. 3.2. In Sect. 3.3, we discuss some aspects related to parallel execution of GAM systems, including the GAM parallel memory model. Finally, in Sect. 3.4, we present the C++ library that we implemented based on the GAM abstract model.

3.1 System Model

A GAM system consists in a set $e_1, \ldots, e_n$ of executors issuing memory operations over a global address space. If a global address is mapped, it points to a memory slot of arbitrary size.

Moreover, each slot is either public or private, according to the associated access capability. A public slot can be accessed by any executor via `load` or `store` operations, although it cannot be updated once a value has been stored into it—i.e., GAM public slots are *single-assignment*. Conversely, a private slot can be accessed via `load` and `store` operations, but only by its *owner*, that is, the executor owning exclusive access capability over the slot.

A capability represents the way in which a given memory slot can be accessed by a given executor. For a public slot, a load-only capability is

Operation	Meaning
`map`	Allocate a slot, either public or private
`unmap`	Free a slot
`load`	Retrieve the value stored in the slot
`store`	Store a value into the slot
`pass`	Transfer the slot capability to another executor
`publish`	Make the (private) slot public

TABLE 3.1: GAM memory operations.

associated to some executors, whereas no executor has store capability on the slot. Conversely, for a private slot, a load-store capability is associated to exactly one executor, that owns exclusive access to the slot.

In addition to memory access operations, executors may issue operations for managing capabilities, namely, `pass` and `publish` When a slot is passed from an executor e_i to another executor e_j, the associated capability is transferred to e_j. In the case of a public slot, e_i also retains the read-only capability, whereas in the case of a private slot, the read-write capability is lost by e_i. Finally, a private slot may be published to make it public, whereas the converse operation is not possible. Table 3.1 summarizes the operations that may be issued by GAM executors.

We proceed by describing step by step a simple execution of a GAM system (Sect. 3.1.1) and by informally comparing GAM systems with those based on cache coherence (Sect. 3.1.2).

3.1.1 Journey of a Global Memory Slot

At the beginning of the system execution, all global memory slots are unmapped, thus they do not contain any valid information.

When an executor e_i issue a `map` operation for a *public* slot, a global address γ pointing to a slot of suitable size is made visible to (only) e_i. At this point, any operation involving address γ issued by an executor $e_j \neq e_i$ is not allowed, since e_j has no capability over γ. Also, any `load` issued by e_i to the mapped slot would return an undefined result until e_i issues a `store` to it, that assigns a value v to the slot. Once v has been stored, no more `store` operations to γ are allowed. Instead, e_i may `pass` the slot to any other executor e_j, giving to e_j both visibility and read-only capability over the slot, while e_i retains the same capability itself. Thereafter, the slot may spread over executors by being passed pairwise. Any `load` operation issued by an executor that has read-only capability over the slot will return the value v that was stored by e_i into the slot.

Let us now consider the case of e_i mapping a global address a' to a *private* slot. Similarly to the public slot case, any `load` issued by e_i to the mapped slot would return an undefined result until e_i issues a `store` to it. But, differently from the public case, e_i may keep updating and reading the slot arbitrarily since it has (exclusive) *read-write* capability over it. If e_i passes the slot to another executor e_j, the capability is lost by e_i and gained by e_j, that may arbitrarily access the slot until it passes it to some other executor. At some point, the slot may be converted to a public one by its owner, by means of the `publish` operation.

A slot, either public or private, may be finally released by means of the `unmap` operation. Upon un-mapping, the global address γ associated to the slot is freed and may be reused for mapping a brand new slot.

3.1.2 Comparison with Cache-Coherent Systems

By considering a generic shared-memory multi-processor system, we may set an analogy between GAM executors accessing global slots and processors accessing memory locations through a caching memory system, endowed with a cache-coherence protocol.

In this setting, the concept of capability associated to each global memory slot resembles that of state associated to each cache line. For instance, in the simple Modified Shared Invalid (MSI) protocol, a cache line is either modified, shared, or invalid. Shared lines resemble public slots in that they can be accessed in a read-only fashion by any cache (executor in GAM) with no need for coordination. Modified lines resemble private slots in that only a specific executor has the exclusive responsibility over the most recent value for that cache line (slot in GAM).

Despite the depicted similarities, a GAM system differs fundamentally from any cache-coherent system for at least two reasons. First, not all memory operations are allowed on a given global slot at a given time instant (e.g., a private slot may be neither read nor written by an executor other than its owner), whereas shared-memory models allow arbitrary access to any memory location by any processor. Second, capabilities are managed explicitly by the executors via special memory operations, namely `pass` and `publish`, whereas in cache-coherent memory systems such aspects are hidden in the cache-coherence protocol.

3.2 Operational Semantics

In this section, we characterize the semantics of a GAM system from the perspective of its global memory component. With this abstraction, a system execution is totally described by the effects it produces over the global memory, which we refer to as *the memory evolution of the system*. We represent such evolutions in terms of the Labeled Transition System (LTS) formalism.

Formally, a LTS is a tuple (S, Λ, T), where S is the set of system *states*, Λ is the set of transition labels and $T \subseteq S \times \Lambda \times S$ is the set of labeled transitions. Each transition (s, λ, s'), causes the system to change its state from s to s' due to λ.

We rely on the notion of *trace* to denote a sequence of transitions within an evolution. Formally, a trace θ is a sequence from $\mathcal{T}^*$. In our setting, we refer to LTS system states as *memory states*, since they represent the state of the global memory component of the system at hand. Similarly, we refer to LTS transitions as memory transitions.

We formalize memory states and transitions in Sects. 3.2.1 and 3.2.2, respectively.

3.2.1 Memory States

In our setting, a system state is a mapping from addresses to associated contents and capabilities. To formalize the transition rules, we use the following atomic domains :

- $\gamma \in \Gamma$: the global addresses;
- $C = \{\texttt{public}, \texttt{private}\}$: the allowed capabilities;
- $d \in D$: the generic values that can be stored in memory;
- $E = \{e_1, \ldots, e_n\}$: the executors issuing memory operations over the global memory system.

The global state of the memory, denoted by s, is given by a function having the following signature, where the first component indicates the capability, the second indicates the associated (data) content, and the third indicates the executors having the capability for that address:

$$s : \Gamma \nrightarrow (C, D, \mathcal{P}_{\geq 1}(E))$$

We also make use of the following syntactic shortcuts to characterize the memory states:

- $s(\gamma) = \bot$ indicates that γ is not part of s's domain;
- $s[\gamma \mapsto s']$ denotes the state built from s and modifying solely γ as defined by s';
- $v(s, \gamma)$ denotes the value stored into γ at state s, i.e., $\pi_2\ s(\gamma)$ if $s(\gamma) \neq \bot$, otherwise $\bot$.[1]

3.2.2 Memory Transitions

A GAM system evolves with respect to its global memory whenever executors issue operations over the global memory. We map memory operations to (memory) transition labels, therefore successive operations identify a trace, that is, a memory evolution. Namely, we map a memory operation issued by executor e_i to a transition label of the following form, where the arguments and return sections depend on the specific operation:

$$\lambda = [e_i]\ \text{operation}\ \langle \text{arguments} \rangle\ \langle \text{return} \rangle$$

Table 3.2 summarizes the meaning of arguments and return values for each operation when mapped to LTS transition labels.

We proceed by introducing the rules that define which transitions are allowed to occur within the memory evolution of a GAM system. We represent transition rules in the standard form where premises are expressed in terms of predicate on the initial state and conclusions are valid transitions, represented as $s \xrightarrow{\lambda} s'$, which is syntactically equivalent to (s, λ, s'). Transition rules are described in Figure 3.1.

[1] In general, given a tuple $t = (t_1, t_2, \ldots, t_k)$, $\pi_i\ t = t_i$, for i such that $1 \leq i \leq k$.

operation	arguments	return
`map`	`public/private` capability	allocated address
`unmap`	address to un-map	-
`load`	address to read	stored value
`store`	address and value to store	-
`pass`	address to pass and target executor	-
`publish`	address to publish	-

TABLE 3.2: Mapping of memory operations to LTS transition labels.

Upon a `map` operation, yielding either a map-public or map-private transition, a new global memory address is mapped and made visible to the issuing executor with proper capability. The operation returns the mapped address, that must be non-mapped in the original state s.

Dually, an `unmap` operation, yielding an unmap transition, frees the argument global address so that it can be safely reused. Note that the unmapping can be performed by any executor that holds a capability over the argument address.

The `publish` operation, yielding a publish transition, simply casts from private to public the capability associated to the argument address, provided the address is owned by the issuing executor in the original state.

Capabilities are exchanged through the executors by means of `pass` operations, yielding either pass-public or pass-private transitions, depending on the issuing executor's capability, in the original state, associated to the address being passed. Moreover, in case of pass-private transition, the issuing executor loses its access permissions over the address, so in the resulting state it does not have capability over the address anymore.

Values are stored to global memory slots by means of `store` operations, yielding either store-public or store-private transitions. Storing a value to a public address is allowed only once, as expressed in the premise of the store-public transition, thus requiring that the address is mapped to the undefined value by the memory function. Moreover, as an additional constraint, the store-public transition must occur before the slot is ever pushed, i.e., when there is a single executor having an associated capability. Conversely, a private address may be freely updated at any time by its owner.

Finally, when an executor issues a `load` operation to a memory slot—over which it has an associated capability—and yielding a load transition, it gets in return the value that has been stored most recently to the slot. Since the load transition is not producing any visible effect over the memory state, the resulting state is the same as the original state.

In the following, we informally say that γ is public (resp. private) in state s, defined as $\alpha(s, \gamma) = \texttt{public}$ (resp. `private`) where α, the *access level*, is given by the following function:

Definition 1. *Given a global address γ and a state s, the* access level *for γ in s, denoted as $\alpha(s, \gamma)$, is:*

$$\alpha(s,\gamma) = \begin{cases} \bot & \text{if } s(\gamma) = \bot \\ \texttt{public} & \text{if } \pi_1\ s(\gamma) = \texttt{public} \\ \texttt{private} & \text{if } \pi_1\ s(\gamma) = \texttt{private} \end{cases}$$

From the above rules, it is straightforward to show that, among others, the following properties hold, for any state s:

- A mapped global address γ is either public *for a non-empty set of executors* or private *for a single executor*.
- A mapped global address γ that is public for a non-singleton set of executors has been stored into ($\neq \bot$).

Finally, we formalize the notion of *ownership* for private addresses.

Definition 2. *Given a global address γ and a state s such that γ is private in s, the* owner *of γ in s is the executor $e_i \in E$ such that:*

$$\pi_3\ s(\gamma) = \{e_i\}$$

From the properties introduced above, we know that, for any state, an owner exists for any private address.

3.2.3 State Machine Representation

In Sect. 3.1.1, we informally described a simple execution of a GAM system, by following the journey of a memory slot in both public and private cases. Fig. 3.2 illustrates a graphic version of the journey, in terms of state machine diagram, a formalism syntactically equivalent to the presented LTS. In particular, each state (i.e., a node in the diagram) is labeled with a compact representation of the memory state function (cf. Sect. 3.2.1), whereas each transition (i.e., an edge in the diagram) is labeled with a memory transition (cf. Sect. 3.2.2).

Fig. 3.2a represents the public pointer case. From the initial state, in which all addresses in Γ are unmapped, a map-public transition makes the system evolve into a state in which the returned address γ is mapped to a memory slot, with undefined stored value and public capability associated to the issuing executor e_i. A store-public transition changes the stored value to the argument value d, also prohibiting any further store-public transition to be observed on γ. Finally, a pass-public transition adds the argument executor e_j to the set of executors holding load-only capability over γ.

Fig. 3.2b represents the private pointer case. With respect to the previous case, multiple store-private transitions can be issued on the same address γ. Finally, a publish transition converts the capability over γ from load-store to load-only.

As for load transitions, they are represented as loops, since they do not induce any change to the memory state.

$$\frac{s(\gamma) = \bot}{s \xrightarrow{[e_i]\ \text{map}\ \langle\text{public}\rangle\ \langle\gamma\rangle} s[\gamma \mapsto (\texttt{public}, \bot, \{e_i\})]}\ \text{map-public}$$

$$\frac{s(\gamma) = \bot}{s \xrightarrow{[e_i]\ \text{map}\ \langle\text{private}\rangle\ \langle\gamma\rangle} s[\gamma \mapsto (\texttt{private}, \bot, \{e_i\})]}\ \text{map-private}$$

$$\frac{s(\gamma) = (c, d, e) \wedge e_i \in e}{s \xrightarrow{[e_i]\ \text{unmap}\ \langle\gamma\rangle} s[\gamma \mapsto \bot]}\ \text{unmap}$$

$$\frac{s(\gamma) = (\texttt{private}, d, \{e_i\})}{s \xrightarrow{[e_i]\ \text{publish}\ \langle\gamma\rangle} s[\gamma \mapsto (\texttt{public}, d, \{e_i\})]}\ \text{publish}$$

$$\frac{s(\gamma) = (\texttt{public}, d, e) \wedge d \neq \bot \wedge e_i \in e \wedge e_j \notin e}{s \xrightarrow{[e_i]\ \text{pass}\ \langle\gamma, e_j\rangle} s[\gamma \mapsto (\texttt{public}, d, e \cup \{e_j\})]}\ \text{pass-public}$$

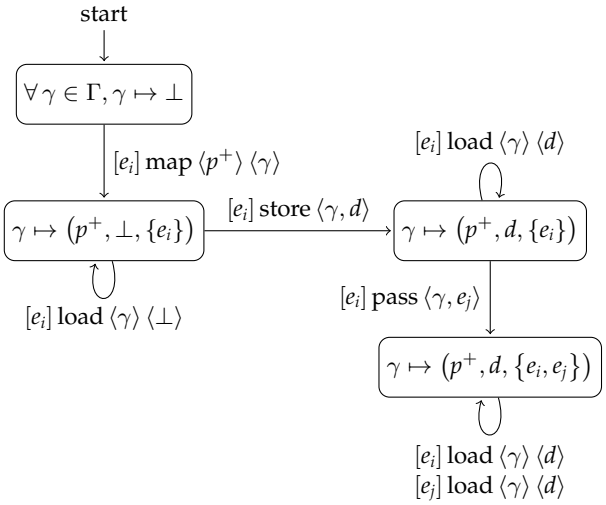
$$\frac{s(\gamma) = (\texttt{private}, d, \{e_i\})}{s \xrightarrow{[e_i]\ \text{pass}\ \langle\gamma, e_j\rangle} s[\gamma \mapsto (\texttt{private}, d, \{e_j\})]}\ \text{pass-private}$$

$$\frac{s(\gamma) = (\texttt{public}, \bot, \{e_i\})}{s \xrightarrow{[e_i]\ \text{store}\ \langle\gamma, d\rangle} s[\gamma \mapsto (\texttt{public}, d, \{e_i\})]}\ \text{store-public}$$

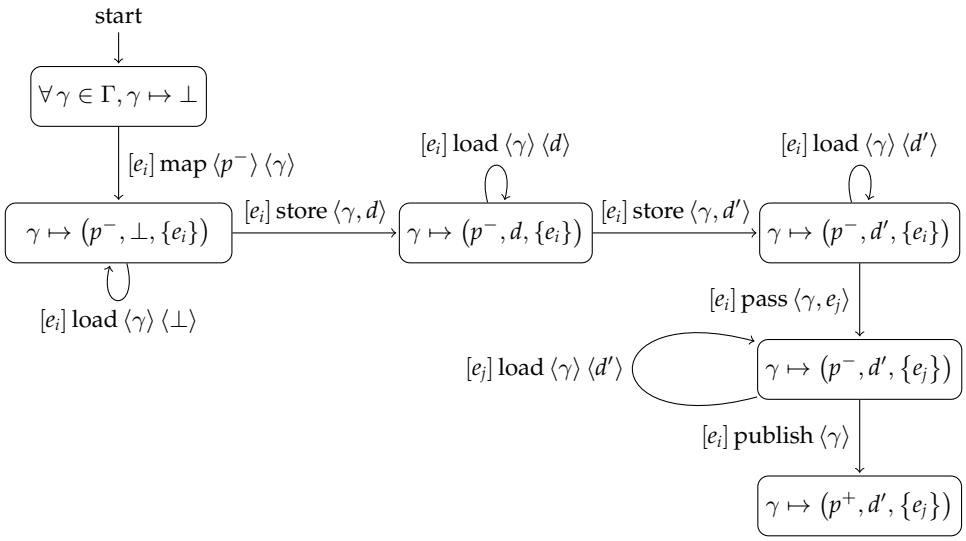
$$\frac{s(\gamma) = (\texttt{private}, d', \{e_i\})}{s \xrightarrow{[e_i]\ \text{store}\ \langle\gamma, d\rangle} s[\gamma \mapsto (\texttt{private}, d, \{e_i\})]}\ \text{store-private}$$

$$\frac{s(\gamma) = (c, d, e) \wedge e_i \in e}{s \xrightarrow{[e_i]\ \text{load}\ \langle\gamma\rangle\ \langle d\rangle} s}\ \text{load}$$

FIGURE 3.1: Memory semantics rule.



(A) Public slot.



(B) Private slot.

FIGURE 3.2: State machines for the journals of memory slots from Sect. 3.1.1. For brevity, we denote `public` and `private` capabilities as p^+ and p^-, respectively. Diagrams evolve horizontally when the state change concerns the stored value, whereas they evolve vertically when the state change concerns the capability.

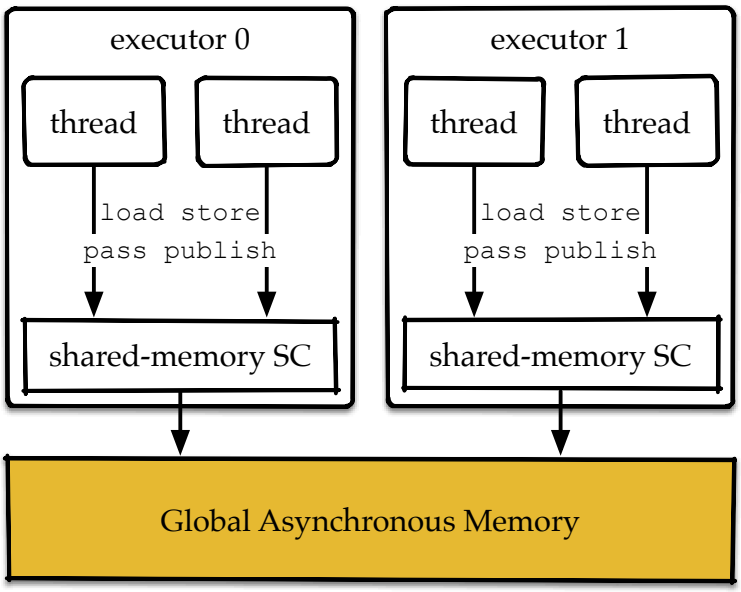


FIGURE 3.3: A simple GAM system composed of two executors with two threads each. For each executor, a logical SC box indicates that SC is provided by the multi-thread shared-memory model within the executor.

3.3 Parallelism

In the previous section, we proposed a semantics for GAM systems in terms of LTS traces, in which all the memory operations issued by executors are linearized. Therefore, according to this abstraction, within a valid execution, all executors agree on a global order over the visible effects on memory. In this section, we evaluate the impact of such a strong abstraction over the available parallelism. More specifically, we show how parallelism can be "neutralized" even in presence of multi-threaded executors.

Figure 3.3 illustrates a simple system, as executed by a hypothetical GAM implementation, composed of two executors, each internally containing two threads. As can be visualized in the example, parallelism with respect to the global memory is exhibited by a system at two levels, namely, within each executor and between executors. We discuss the two forms of parallelism in Sects. 3.3.1 and 3.3.2, respectively. In Sect. 3.3.3, we put all together by defining the parallel memory model for GAM programs.

3.3.1 Intra-Executor Parallelism

The intra-executor form of parallelism arises as different threads may have simultaneous interactions with the global memory. For the sake of simplicity, in the following we avoid targeting this aspect by assuming single-threaded executors. Since in this case all the global memory operations from an executor are issued sequentially, their effects are already linearized into a global order. Moreover, such order respects the order between operations as specified within the program being executed (i.e., the program order).

Nevertheless, it would be straightforward to fulfill the same requirements while retaining multi-threading by integrating the global memory operations into some well-studied multi-threaded memory model. Namely, we would need a multi-threaded memory model providing SC (that we briefly recap in Sect. 3.3.3) as suggested by the SC-like box in Figure 3.3. For instance, SC is guaranteed for a well-formed subset of C++ and Java multi-threaded programs through their respective memory models [38, 99]. In particular, SC is guaranteed for race-free programs, where a race is defined as a situation in which the same memory location may be read and written concurrently by different threads.

In order to integrate GAM operations into one of the mentioned shared-memory models, GAM `load` and `store` operations may be treated as their shared-memory counterparts, thus possibly leading to races unless they are properly synchronized by some special memory operations. From the perspective of the effects over the global memory, all the other operations may be treated as shared-memory `load` operations, since they do not modify the memory map for their argument global addresses. Therefore, given a GAM `store` operation and another operation of any kind targeting the same global address, they must be properly synchronized in order to maintain SC.

We remark that SC does not implies correctness with respect to the memory transition rules defined in Figure 3.1. Indeed, SC only allows to regard executions as LTS traces ordered according to the program order, whereas the memory rules discriminate valid traces from invalid ones.

3.3.2 Inter-Executor Parallelism

The inter-executor form of parallelism arises as different executors may have simultaneous interactions with the global memory.

The key principle underlying the proposed memory semantics is that the effects over the global memory can be induced *locally* by each executor. This can be easily visualized from the memory rules in Figure 3.1 for all operations except for `load` and `store`. For instance, `map` and `publish` operations induce a local update to the capability function for the issuing executor. Similarly, performing a `pass` on an address induces only local updates to the capability functions for the involved executors.

The `load` and `store` operations apparently require more effort. As a side effect of abstracting executions into (linear) LTS traces, the proposed memory semantics also provides a strong notion of store atomicity [121]: a `store` issued by an executor is logically seen by all executors at once.[2] However, if we consider for instance the case of an executor e_i performing a `store` to a private slot γ, only the executor e_j that will get the read-write capability over γ (upon a `pass` by e_i) will have access to the stored value. Therefore, for a valid implementation it would be sufficient to guarantee that the effect of the `store` is visible to e_j once γ has been passed, in order to maintain store atomicity. The same reasoning applies also in case of passing public slots, which is an even simpler case since their value is assigned once and never changed, allowing them to be cached.

[2] As mentioned in [121], a weaker notion of store atomicity is generally adopted, requiring a store issued by an executor is logically seen by all *other* executors at once.

In the context of defining the parallel memory behavior, enabling the effects on memory to be localized allows to release any restriction on the parallelism between executors. We remark that such a simplification is driven by the definition of memory rules, that prevents any form of inter-executor conflicting memory accesses [38], thus excluding data races between different executors.

3.3.3 Parallel Memory Model

From the memory semantics defined in terms of LTS traces, we extract an abstract memory model that specifies the behavior of multi-executor programs attached to the global memory. To this aim, we proceed along the same line as the discussion on shared-memory memory models by Sorin *et al.* [121], addressing separately the aspects of coherence and consistency.

Coherence

We deal with coherence by adapting the formulation of Sorin *et al.* [121] (cf. Sect. 2.3.1). We define a GAM system as *coherent* if and only if it respects the following invariants:

- The SWMR invariant states that, for any given global memory slot, it can be divided up into epochs such that, in each epoch, either a single executor has read-write access or some number of executors (possibly zero) have read-only access;
- The DV invariant states that the value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

From the memory rules in Figure 3.1, we discuss how the memory semantics deals with coherence. A public address γ is mapped in an initial epoch in which only the executor that mapped γ owns both visibility and read-write capability over γ. After it has been stored, it passes to an epoch in which all the executors that have visibility on γ also have read-only capability over γ. As for a private address γ, each time it gets passed by an executor e_i to another executor e_j, it passes from an epoch in which e_i owns exclusive read-write capability to another epoch in which it is owned by e_j. Therefore, the SWMR invariant is respected.

As for the DV invariant, it holds trivially: at the start of an epoch identified by a transition for the address γ, the value of the memory location (i.e., $v(s, \gamma)$ in Figure 3.1) is the same as the value of the memory location at the end of its last read-write epoch, identified by the last store transition for γ.

Therefore, the memory semantics we provided for GAM respects both invariants. Note that we do not need any additional effort in order to define memory epochs, since they are already provided by the linear nature of the LTS traces.

Sequential Consistency

The problem of defining a memory model for GAM programs is more general. By adapting the definition of Sorin *et al.* [121], a memory (consistency) model is a specification of the allowed behavior of multi-executor GAM

programs executing with global memory. In general, a memory consistency model gives rules that partition executions into valid and invalid ones with respect to the model, that in turn partitions implementations.

The simplest memory model is sequential consistency, which was first formalized by Lamport [95]. By adapting the original definition, we define a GAM implementation to respect SC if the result of any execution is the same as if the operations of all executors were executed in some sequential order, and the operations of each individual executor appear in this sequence in the order specified by its program. This (total) order of operations is called *memory order* and in SC it respects the program order of each executor.

The most relevant consequence of relying on the proposed semantics, based on linear traces, is that it allows to reason about consistency in a straightforward way, as we show in the following.

We first prove that any GAM execution fulfills the first requirement for SC, that is, it provides a global order for `load` and `store` memory operations. To this aim, given an execution, it is sufficient to consider its trace θ (see Section 3.2) and remove all transitions except for those involving `load` and `store` operations. From the rules in figure 3.1, the resulting trace $\theta' \subseteq \theta$ is a sequence of transitions with either of the following two forms, where the capability c in the second rule is either `private` or `public`:

$$s \xrightarrow{[e_i]\ \text{load}\ \langle\gamma\rangle\ \langle v(s,\gamma)\rangle} s$$

$$s \xrightarrow{[e_i]\ \text{store}\ \langle\gamma,d\rangle} s[\gamma \mapsto (c, d, \{e_i\})]$$

By mapping θ' to a plain sequence of `load` and `store` operations over the shared address space, we obtain an execution that defines by construction a global order over these operations.

As for the requirement about respecting program orders, we already discussed in Section 3.3.1 how program orders can be honored for each executor by a sequential execution, in case of single-threaded executors. Since the interleaving of ordered sequences produces a sequence that respects each of the components order, any GAM execution in which each executor provides internal SC is itself sequentially consistent. Therefore, the GAM implementation we depicted in Sections 3.3.1 and 3.3.2 is SC, at least in the case of single-threaded executors.

In the case of multi-threaded executors, each executing according to some shared-memory SC memory consistency model, the same SC model would be inherited by the whole GAM implementation. For instance, a GAM implementation with C++ multi-threaded executors would be SC for data-race-free programs, according to the simple extension of the C++ memory model that we depicted in Section 3.3.1.

3.4 C++ Implementation

In this section, we present a C++ implementation that partially realizes the system model introduced in Sect. 3.1. In particular, it respects the semantics presented in Sect. 3.2 and supports the forms of parallelism discussed in Sect. 3.3.

The implementation we present is not intended to be used directly by a user application. Instead, it should be regarded as a low-level software support for higher-level libraries. In this perspective, we remark that the presented implementation is not meant to be neither strictly correct nor complete with respect to the proposed semantics. For instance, as a counterexample for correctness, the implementation allows a malicious user to get around the ownership rules and keep writing to an address after it has been passed to another executor. And as a counterexample for completeness, the implementation does not allow to map an address without storing an initial value into it—as we will see below, the only way of mapping an address is via `mmap_public`, which takes a local pointer and implicitly initializes the content of the address being mapped with the local argument pointer.

We opted for this design choice since it is along the very same lines as the C++ philosophy of keeping the bottom runtime layer as lightweight as possible. For instance, the C++ runtime provides limited support for capturing invalid memory accesses, simply marking as undefined all such programs that violate the memory semantics (e.g., data races or accesses to unallocated memory). Along the same lines, we propose an implementation that, instead of providing a fully featured management and control of each global memory access, simply exhibits undefined behavior in some cases of semantic violation. The simplified rationale behind this choice is freeing the runtime from performing cumbersome sanity checks—thus enabling extra optimizations—by pushing some responsibilities to the programmer over correctness.

According to this approach, in the following chapters we show in a constructive manner that the implementation we propose is powerful enough to serve as baseline for higher-level libraries, where both correctness and completeness issues are progressively hidden. Again, according to the C++ philosophy, we rely on abstraction in form of higher-level APIs (cf. Ch. 4), providing correctness in terms of constructive programming rather than heavyweight runtime.

We proceed in Sect. 3.4.1 by describing at high level the user interaction with the implemented library. Then we shift to the implementation, by showing, in Sect. 3.4.2, the general architecture of the library RTS. Finally, in Sect. 3.4.3, we detail the library API in an analytic manner, providing also some implementation details for each primitive.

3.4.1 Programming Environment

In the proposed implementation, an executor is a process that eventually issues calls to a C++ library that we refer to as *the GAM runtime*. We provide a simple programming environment inspired by SPMD environments (e.g., MPI), in which each process is marked with a unique rank ranging from 0 to $n - 1$, where n is the number of executing processes, that we refer to as the cardinality. The rank and the cardinality may be obtained from the runtime by calling the functions `rank` and `cardinality`, respectively.

On the same line as an MPI-like environment, we provide a simple launcher that reads a topology from a user-provided file and takes care of launching processes, either locally or remotely via `ssh`. The launcher also

```cpp
#include <iostream>
#include <gam.hpp>

// Define each executor's callable.
void c0() { ... }
void c1() { ... }

int main(int argc, char * argv[])
{
    // Print executor rank.
    std::cout << "My rank is " << gam::rank() << std::endl;

    // Execute rank-specific code.
    switch (gam::rank())
    {
    case 0:
        c0(); // Invoke rank 0 callable.
        break;
    case 1:
        c1(); // Invoke rank 1 callable.
        break;
    }

    return 0;
}
```

LISTING 3.1: Skeleton of a minimal GAM application with two executors.

sets some environment variables, so that they can be reached by the GAM runtime during the initialization phase.

The code snippet in Listing 3.1 shows the skeleton of a minimal application with two executors, each printing its rank and executing a rank-specific callable object. The associated program should be launched by a command line similar to the following, where the topology file is a list of network hosts, the binary is reachable at the same path by each host (e.g., as absolute path in a distributed file system) and the `-n` option specifies the cardinality:

```
gamrun -n 2 -f topology.conf /path/to/binary
```

3.4.2 Runtime Architecture

During the execution of a GAM program, each executor process is attached to an instance of a GAM *context* that wraps all runtime components and represents, for each executor, its local interface to the global memory. Each time the executor needs to interact with the global memory, it issues a call to a context function. We discuss the most relevant context functions in Sect. 3.4.3.

Fig. 3.4 illustrates the architecture of the GAM runtime in a scenario with two executors and shows some typical interactions between the user thread and runtime components. In the following, we discuss the meaning of each component.

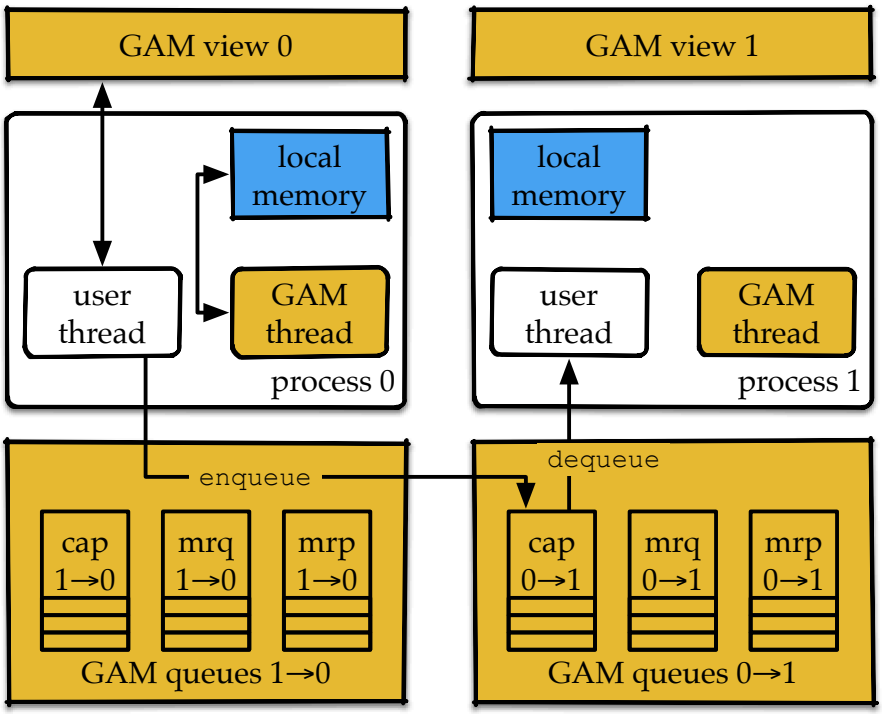


FIGURE 3.4: Architecture of the GAM runtime with only two executors.

View

For each executor e_i, its knowledge about the global memory is managed by a runtime component that we call the GAM *view*. Globally, the views attached to all executors realize a distributed implementation of the memory states, as defined in Sect. 3.2.1. First, a local view maps each global address γ to its respective capability $(\alpha(s, \gamma))$. Moreover, it is responsible for storing and providing the value $v(s, \gamma)$ for such addresses that e_i is the *author*[3] of, in the sense that it performed the most recent `store` operation for γ. Namely, if (and only if) e_i is the author for γ, a *stored* pointer is associated to γ, that points to the local memory slot containing the value $v(s, \gamma)$.

Communication

The memory semantics that we proposed in Sect. 3.3.2 is less expressive than traditional DSM models but allows to minimize the interaction between the executors. As a constructive proof, in Sect. 3.4.3 we present an implementation such that exchanging capabilities and data requires the asynchronous interaction of up to two executors, whereas all other primitives are performed locally by the issuing executor.

The communication between executors is realized by means of asynchronous operations over communication *queues*, as provided by the libfabric library. The runtime associates three queues for each pair of executors, namely a capability queue, a memory request queue and a memory reply

[3]We remark that the authorship concept is irrelevant for the operational semantics formalized in Sect. 3.2, whereas it arises at implementation level.

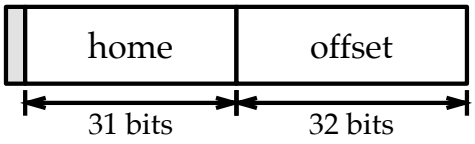


FIGURE 3.5: Layout of a GAM global address in the proposed C++ implementation.

queue, denoted respectively as cap, mrq and mrp in Figure 3.4. For such queues, both directed and undirected popping primitives are provided, realizing respectively pop-from and pop-from-any semantics.

Finally, an additional GAM *thread* is associated to each executor, with the purpose of serving remote load requests.

3.4.3 Primitives

Among the functions provided by each context to the attached executor, we present the most relevant ones with respect to the semantic rules in Fig. 3.1. We proceed in ascending order of complexity, in terms of number of components involved in each function.

Basic Types

A global address is represented by a $64\,\mathrm{bit}$ value, whose bitwise layout is illustrated in Fig. 3.5. To enable the freshness of an address to be guaranteed locally, we partition the global address space into disjoint subspaces by halving each address and encoding the rank of the mapping executor in the most significant region (the home region in figure). Moreover, as in the virtual memory layout provided by many operating systems, we reserve a portion of the address space for storing arbitrary values rather than memory locations to be mapped. Namely, any address for which the most significant bit (highlighted in gray in figure) is set to 1 is non-mappable, thus resulting in 2^{63} mappable addresses partitioned among 2^{31} executors (each providing 2^{32} addresses) plus 2^{63} non-mappable values. As we show in Ch. 5, non-mappable values are meant to be exploited by applications on top of the GAM stack.

In this setting, the implementation we propose is based on two fundamental types, representing, respectively, global addresses and executor identifiers:

```
class GlobalPointer {
  /* ... */
  uint64_t descriptor;
};

typedef uint32_t executor_id;
```

Mapping

Global memory slots can be mapped by means of the following primitives, for public and private slots, respectively:

```
template<class T, typename Deleter>
GlobalPointer mmap_public(T *lp, Deleter d);

template<class T, typename Deleter>
GlobalPointer mmap_private(T *lp, Deleter d);
```

Both functions take as argument a local memory pointer `lp` whose content is used to initialize the value associated to the global address γ being mapped. Under the hood, both functions perform the very same actions with respect to the runtime when invoked by an executor e_i, namely:

1. Find a fresh global address γ to map;
2. Bind the memory value $v(s, \gamma)$ to (the content of) `lp`;
3. Set the authorship of γ to e_i;
4. Set the capability ($\alpha(s, \gamma)$) to either `public` or `private`.

All the above actions can be performed locally by e_i: after a fresh address has been selected (item 1), an implicit store is performed (item 2) by setting the stored pointer for γ to `lp` within the view associated to e_i; then, both authorship and capability for γ are updated (items 3 and 4, respectively), again within the view associated to e_i.

Upon mapping, the responsibility over `lp` is logically transferred to the runtime, similarly to what happens for C++ smart pointers. Therefore, again in line with smart pointers, the local pointer should not be accessed anymore outside the runtime to ensure correctness with respect to the GAM semantics.

Both mapping primitives also take as input a destructor object that will be used to free the memory associated to `lp` when γ is un-mapped by calling the function:

```
void unmap(const GlobalPointer &)
```

In the implementation we propose, un-mapping a slot amounts to clearing the corresponding entry from the view associated to the issuing executor.

Upon un-mapping, according to the transition rules in Fig. 3.1, any access to the address γ associated to the un-mapped slot results in invalid behavior, unless another slot is mapped to γ due to address reusing. However, as we anticipated at the beginning of this section, the implementation we propose does not provide any mechanism for preventing such undefined behaviors.

Passing Capabilities

The following primitives implement the `pass` operation for exchanging capabilities between executors, where the two variants for `pull` serves for pulling from either any executor or a specific executor `from`, respectively:

```
void push(const GlobalPointer &p, const executor_id to);

GlobalPointer pull();
GlobalPointer pull(const executor_id from);
```

Upon pushing, the issuing executor performs the following actions:

1. In the case of a private address, the local view is updated such that the ownership is set to the target executor;

2. A message—including global address, authorship and capability—is pushed to the capability queue (i.e., cap in Fig. 3.4) associated to the target executor.

Dually, upon pulling, the issuing executor (i.e., the target) performs the following actions:

1. A message is popped from one of its capability queues, either a specific one (in case of pulling from a specific executor) or any of them (in case of pulling from any executor);

2. The local view is updated such that the popped address is mapped and all the information match the content of the message.

As for the mapping/un-mapping operations, the actions for pushing and pulling are performed locally by the respective executors.

Note that, although the implementation performs only local changes to the state of the issuing executor, the `push` primitive corresponds itself to the semantic rule for passing (either pass-public or pass-private in Fig. 3.1). Nevertheless, the actual updating regarding the capability at the target side, as required by the semantic rule, is deferred until the capability is retrieved by means of the `pull` primitive.

Note also that pushing a private address may induce some staleness. In general, if a private address γ is pushed from e_i to e_j and then from e_j to e_k, e_i does not get notified about the new ownership, therefore its view contains stale information. Nevertheless, the effect of such staleness is neutralized by the weak sharing semantics of the GAM model, since an executor is prevented from accessing private pointers owned by another executor.

Finally, we remark that the communication schema for passing capabilities is fully asynchronous, since neither the issuing nor the target executor needs to interact with its peer in order to complete the action, as can be visualized from the push and pop edges in Fig. 3.4.

Loading, Storing and Publishing

The proposed implementation does not provide explicit mechanisms for loading and storing data from and to global memory. Instead, it provides an implicit mechanism based on converting global addresses to local pointers.

Public addresses are loaded by means of the following function:

```
template<typename T>
std::shared_ptr<T> local_public(const GlobalPointer &p);
```

This produces a fresh copy of the memory pointed by `p`, allocated from the local memory of the issuing executor.

Note that the return type is a C++ shared pointer, so that the entire lifetime of the resulting copy is managed by the runtime. Producing a new copy upon each access forces a read-only semantics for public pointers, since there is no mechanism to reflect the modifications made to a local copy back to the public address from which the copy was generated. For the same reason, we safely implemented a *caching* infrastructure for public addresses, so that remote interactions between executors for reading contents of public addresses are minimized.

When issued, `local_public` performs the following actions:

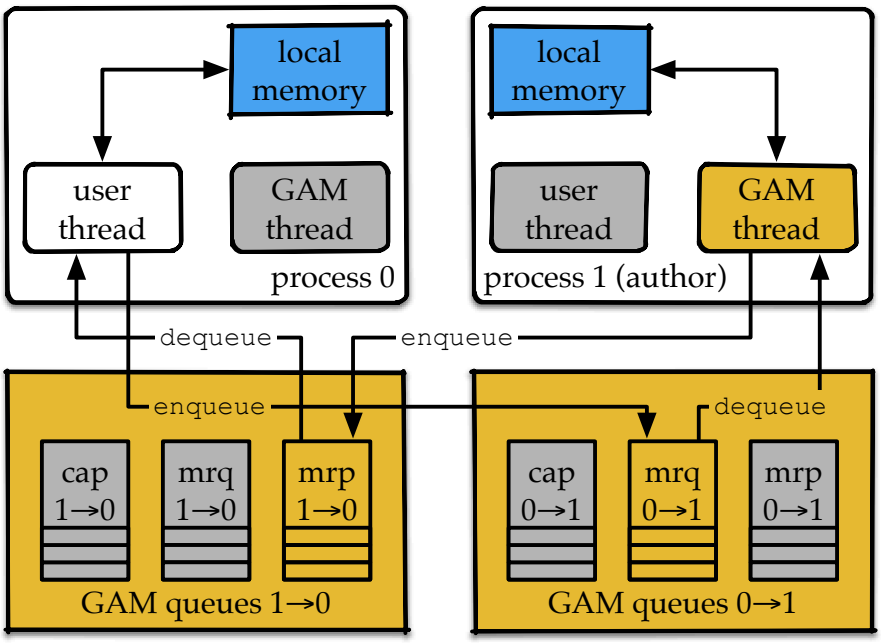


FIGURE 3.6: Runtime actions for remote loads, with only two executors.

1. A memory chunk of `sizeof(T)` bytes is allocated from local memory;
2. Memory pointed by `p` is loaded into the allocated copy:
   - If the issuing executor is also the author, memory is loaded from the stored pointer, as for the load edge in Fig. 3.4;
   - If the issuing executor is not the author but the memory pointed by `p` has already been *cached*, the cached memory is loaded;
   - If memory has not been cached, then a remote load is issued to the author executor and performed with the collaboration of the GAM thread at the author side, as illustrated in Fig. 3.6;
3. A C++ shared pointer is prepared with a custom deleter that reflects the allocation primitive used in item 1.

We remark that the allocated copy is independent from the original global pointer, therefore it may safely be manipulated without compromising the correctness with respect to the GAM semantics.

A similar interface is provided for converting private addresses, where the only difference with respect to the public case is the return type, in this case a raw pointer:

```
template<typename T>
T *local_private(const GlobalPointer &p);
```

According to a zero-copy design principle, the runtime associates a local memory chunk (i.e., the chunk pointed by the stored pointer) to each private address and it keeps such chunk unique across the aggregated space of the local memories of all executors. Indeed, converting a private address γ to a local pointer yields the stored pointer for γ, which allows to arbitrarily read and write γ by accessing the memory chunk associated to γ.

The zero-copy approach also precludes using smart pointers as return values for `local_private`, since the destruction of a memory chunk at some executor side must be deferred until the executor is no more an author for the corresponding address. Such change in the authorship for an address γ is triggered upon receiving a remote load for γ, which from that point may safely un-map its local memory chunk associated to γ, as we explain in Sect. 4.2.

Therefore, when the conversion is issued by the author executor, it simply returns the stored pointer. On the other hand, when the issuing executor is not the author, the following runtime actions are performed:

1. A memory chunk of `sizeof(T)` bytes is allocated from local memory;
2. A remote load (Fig. 3.6) is performed, that also triggers un-mapping at the receiver side;
3. The authorship of γ is set to the issuing executor.

Finally, publishing performs the same actions as converting, plus changing the capability to public after both the ownership and the authorship have been transferred (if needed) to the issuing executor. Publishing a private address is provided by the following function:

```
template<typename T>
void publish(const GlobalPointer &p);
```

Summary

In this chapter, we introduced GAM (Global Asynchronous Memory), a shared-memory model with limited interface (i.e., not a generic load/store) for distributed platforms, providing sequential consistency on top of non-coherent hardware. We presented a C++ API and implementation of GAM, based on libfabric for targeting a variety of large-scale HPC platforms.

Chapter 4

Smart GAM Pointers

In this chapter, we introduce *smart GAM pointers* on top of the GAM model and implementation presented in Chapter 3. Smart GAM pointers are the core components of the API for coding GAM programs. We designed both the API and its implementation as a porting of the C++ smart pointers API to a distributed programming environment.

Following the distinction between C++ shared and unique pointers, we propose two classes of pointers, namely *public* and *private*. Public pointers resemble shared pointers, in the sense that different *copies* of a public pointer, distributed among the executors, share the control over the underlying (public) memory slot. Symmetrically, private pointers resemble unique pointers, in the sense each pointer has exclusive control over the underlying (private) memory slot.

Public pointers provide automatic management of public memory slots on top of a distributed protocol for reference counting. Similarly, private pointers provide management of private memory slots, on top of a distributed protocol for memory releasing. The abstraction induced by automatic global memory management is at the core of the *smartness* provided by smart GAM pointers. We target smartness on two dimensions, each corresponding to a class of memory errors, namely, memory leaks (i.e., unreachable global slots) and dangling pointers (e.g., useless global references). In a sense, we extend the notion of smartness provided by C++ smart pointers to the context of global memory programming.

This chapter proceeds as follows. We present public and private pointers in Sects. 4.1 and 4.2, respectively, while we discuss the aspects related to smartness in Sect. 4.3.

4.1 Public Pointers

We propose *(smart) public pointers* as an abstraction, in terms of C++ template class, for managing public memory slots. They provide automatic memory management for public slots, from their creation to their destruction, which is automatically triggered when the slot is not accessible anymore by any executor. They also emulate raw global pointers, in the sense that they provide functions for seamlessly accessing the managed memory slots.

Under the hood, a public pointer contains a (raw) global pointer that points to a public slot. In cooperation with all the other copies of the public pointer, that are possibly spread among all the executors, the public pointer maintains the control over the contained pointer. To this aim, it maintains a count of the references to its contained pointer. The referenced slot will

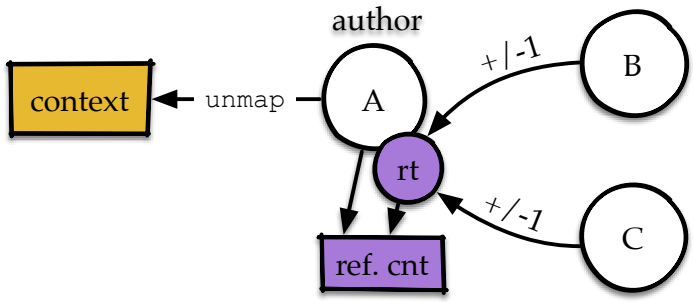


FIGURE 4.1: Schema for the distributed reference counting protocol.

be unmapped, causing the corresponding stored memory to be freed, when and only when all the copies of the public pointer have released the control over the contained pointer. Since multiple copies of a public pointer may exist among the executors, we refer to such counting mechanism as *distributed reference counting*.

We proceed by first presenting an implementation of the distributed reference counting in Sect. 4.1.1. Then we detail the most relevant functions provided by the public pointer API in Sect. 4.1.2. For each function, we present the signature along with a brief description of its implementation. In particular, we detail the interaction between the implementation of the public pointer API and the functions provided by the runtime context (cf. Sect. 3.4.3).

4.1.1 Distributed Reference Counting

In order to support efficient automatic memory management for public slots, we implemented a mechanism for counting references to public slots in a distributed manner. Namely, we defined and implemented a protocol for distributed reference counting, based on the cooperation of all the executors that control each public slot.

Figure 4.1 illustrates the basic schema of the implemented protocol. For each public slot at global address γ, the author executor (A in the figure) associates to γ a reference counter that is initialized to 1 when γ is mapped. The counter is either incremented or decremented each time a reference to γ is created or dropped, respectively, on any executor. According to the distributed protocol, non-author executors (B and C in the figure) send increment and decrement requests to the author, that processes such requests by means of a service thread (rt in the figure) running along with the other executor threads.

Since the author too may create and drop references to a public slot, both the service thread and the other author threads may try to update a reference counter concurrently. Therefore, we implemented reference counters as *atomic* values that, according to the C++ memory model [38], provide automatic support for concurrent accesses. Moreover, we designed the communication between (non-author) executors and service threads according to the same approach described in Sect. 3.4 for remote loads. Namely, a libfabric queue is associated to each service thread and the communication

is carried in terms of asynchronous push and pop operations over such queues.

From the performance perspective, the discussed reference counting protocol potentially induces non-negligible overhead, depending on the frequency at which protocol requests (i.e., increment or decrement) are issued to service threads. However, since automatic memory releasing can be considered a *low-priority* task with respect to the overall GAM system, we configure the service threads for relying on the *blocking pop* libfabric primitive for consuming the incoming requests. This enables the operating system of each executor to suspend the service thread until some request is delivered to its queue, thus releasing the associated resources and making them available for tasks with higher priority.

More generally, the performance impact of distributed reference counting can be modulated by setting the priority of the service threads. An extreme scenario is observed with maximum priority, in which protocol requests are served eagerly (thus minimizing the overall memory footprint) at the price of maximum performance impact. The opposite extreme scenario is observed with null priority, in which protocol requests are not served at all (thus maximizing the overall memory footprint) and the impact on performance is neutralized.

From the considerations above, it follows that the impact of distributed reference counting on system *scalability* depends on both the priority assigned to service threads and the number of public pointers referencing the same global address γ. Indeed, each public pointer to γ issues at least one decrement request (upon destruction), in addition to increment requests generated upon copying, directed to the same service thread.

4.1.2 API

A public pointer is represented by an object of type `public_ptr<T>`, where the template parameter `T` represents the type of the referenced memory content. Public pointers can be created by means of the following constructor:

```
template<typename Deleter>
public_ptr(T * const, Deleter);
```

The constructor first maps a public slot by calling the `mmap_public` function, then stores the returned global pointer as its contained pointer. It also initializes the reference counter for the mapped slot. We recall that the argument local pointer and delete are used respectively as stored pointer and resource release callback.

In addition to explicit construction, a public pointer can be created by means of the generator function:

```
template<typename T, typename... Args>
public_ptr<T> make_public(Args&&... args);
```

This function internally constructs a `T`-typed object by invoking the `T` constructor with parameters `args`. Then it invokes the `public_ptr` constructor passing as arguments the address of the constructed object and a destructor that matches the allocation primitive employed for the construction. With respect to the explicit constructor case, the generator function is more abstract since it does not require the user to encode any knowledge about the allocation of the local memory referred by the global slots. Note that,

since C++ 17, class template arguments can be deduced from the type of initializers (e.g., constructors). Therefore, the `make_public` function could be rewritten as a template constructor.

A public pointer can be copied, producing a new public pointer that is an additional controller for the contained pointer. Copying a pointer also leads to incrementing its reference counter, either locally or, when the copy is generated by an executor different from the author, remotely by sending an increment request to the author, as shown in Figure 4.1. Similarly, constructing a public pointer by means of the move constructor produces a new controller for the contained pointer. However, moving does not require modifying the reference counter since, in compliance with the C++ move semantics, the argument public pointer being moved gets emptied by the constructor, thus it drops the control over the contained pointer.

We also consider pulling as a form of creation, since it adds the issuing executor to the set of controlling executors for the pulled slot by creating a new public pointer:

```
template<typename T>
public_ptr<T> pull_public() noexcept;

template<typename T>
public_ptr<T> pull_public(executor_id from);
```

The two variants reflect the variants of the corresponding context function for pulling from any or from a specific executor, respectively. However, differently from the previous cases, pulling is agnostic with respect to reference counting, as it is is instead managed on the pushing side.

Finally, a public pointer may be constructed by converting a private smart pointer:

```
template<typename T>
explicit public_ptr(private_ptr<T> &&p) noexcept;
```

Such conversion is meant to transfer the control over the memory pointed by `p` to the public pointer being constructed, thus also making it controlled cooperatively instead of exclusively by the argument private pointer. In terms of reference counting, this leads to a new reference counter being associated to the contained pointer and initialized, whereas the argument private pointer is released (cf. Sect. 4.2.3) to drop the control over the underlying memory slot. In terms of GAM runtime, the conversion is realized by calling the `publish` function.

Dually to construction, a public pointer is destructed when its destructor is invoked, either by the C++ runtime or explicitly. Destructing a public pointer represents dropping the control over the contained pointer, therefore the associated reference counter has to be decremented, either locally or remotely. Decrementing the reference counter is provided by the `reset` function, that also resets the contained pointer for the public pointer. When such decrement is triggered by the last existing public pointer controlling its contained pointer, eventually the author will set the reference counter to zero. Then it finally un-maps the referenced slot by calling the `unmap` function, which internally calls the destructor for `T` specified at construction time. Destructing a public pointer is not the only operation that leads to resetting it. For instance, assigning a public pointer `q` to a variable `p` of the same type (i.e., overwrite) makes `p` replace its contained pointer with

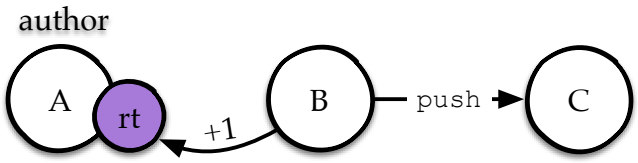


FIGURE 4.2: Reference counting protocol in case of pushing a public pointer from a non-author executor.

the one controlled by `q`, leading `p` to implicitly drop the control over the previously referenced slot.

A public pointer can be converted to a local pointer, for instance to enable its value to be processed by regular C++ code:

```
std::shared_ptr<T> local();
```

Internally, this simply calls the corresponding runtime function, passing as argument the contained pointer. As we discussed in Sect. 3.4.3, a copy of the memory referenced by the contained pointer is generated and returned in form of plain C++ shared pointer, in order to enforce the read-only semantics for public slots.

Finally, a public pointer can be pushed to another executor to diffuse the associated read-only capability:

```
void push(executor_id to);
```

Figure 4.2 illustrates the case of an executor B pushing to another executor C a public pointer whose author is a third executor A. As can be visualized from the example, pushing a public pointer also increments the reference counter associated to the contained pointer, since the pushed slot will be wrapped into a new public pointer when pulled. Instead of deferring the increment at pulling time, we perform the increment at pushing time to avoid the possibility for the counter to reach zero (thus causing the slot to be un-mapped) before the slot is pulled.

Table 4.1 summarizes the most relevant functions provided by the public pointers API. For each function, the table reports both the corresponding runtime function that is executed under the hood and its effect with respect to distributed reference counting.

4.2 Private Pointers

In the same vein as in the previous section, we propose *(smart) private pointers* as an abstraction for managing private memory slots, providing automatic memory management and emulation of raw pointers.

Differently from public pointers, a private pointer has exclusive control over the contained pointer: once it takes control, it manages the pointed slot by becoming responsible for its deletion at some point. Indeed, a private pointer automatically deletes the associated slot as soon as it loses the control over the slot (e.g., by destruction or assignment), unless the control is passed to some other pointer. We implemented a simple distributed protocol to support the depicted mechanism for memory releasing.

Two types of private pointers are provided by the API, namely, *global* and *local*, in form of different C++ classes. At any time, a private pointer

Function	Back-end GAM primitive	Reference counting
constructor	`mmap_global`	initialization
copy cons.	-	increment
move cons.	-	-
from-private cons.	`publish`	init.
`make_public`	`mmap_global`	init.
`pull_public`	`pull`	-
destructor	`unmap`[a]	decrement
copy assignment	`unmap`[a]	dec. + inc.
move assignment	`unmap`[a]	dec.
`local`	`local`	-
`push`	`push`	inc.

[a]Only if the pointer is the last remaining for the controlled slot.

TABLE 4.1: Public pointers API.

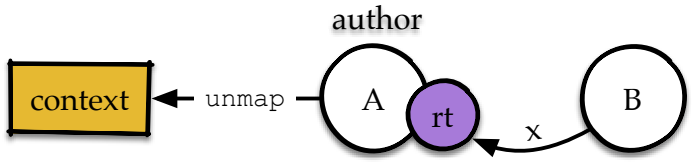


FIGURE 4.3: Schema of the protocol for distributed memory releasing.

assumes either its global or local form, both having exclusive control of the same private slot.

We proceed by first presenting the distributed protocol for memory releasing in Sect. 4.2.1. Then, we motivate and present the global and local flavors of private pointers in Sect. 4.2.2. Finally, we detail the most relevant functions provided by the private pointer API in Sect. 4.2.3.

4.2.1 Distributed Memory Releasing

In order to support efficient automatic memory management for private slots, we implemented a simple protocol for distributed memory releasing, involving up to two executors, namely the author of the slot and its owner. Each private slot mapped at global address γ is controlled by exactly one executor, by means of a private pointer object. If the executor drops the control without passing it to another executor, then the slot can safely be un-mapped, since at that point it becomes definitively unreachable.

Figure 4.3 illustrates the setting for the implemented protocol. We distinguish two cases, depending on the identity of the executor that definitively drops the control over a slot it controls by means of a private pointer. In the simplest case, the slot is dropped by its author, which simply requires the author to call the runtime function `unmap` (cf. Sect. 3.4.3) to trigger the un-mapping of the slot. In the other case, the slot is dropped by a non-author executor (B in the figure), which requires such executor to notify the

author by pushing a release request (the x edge in the figure) to a queue attached to the author. Once the author receives the notification, it proceeds with the un-mapping by calling the `unmap` runtime function. The processing of release requests at the author side is implemented in the same way as in distributed reference counting (cf. Sect. 4.1.1): an auxiliary thread (rt in the figure) pops requests from a libfabric queue by means of blocking primitives.

From the considerations in Sect. 4.1.1, we can conclude that private pointers induce less performance impact than public pointers. Indeed, each private pointer generates at most one release request, upon destruction by a non-author executor. Therefore, the only scenario in which the distributed memory releasing protocol may impact scalability consists in multiple executors issuing release requests to the same executor. This scenario results, for instance, from a GAM program in which a single executor generates a stream of values and distribute them to a set of downstream executors, as in the *farm* pattern (cf. Sect.5.3.2).

4.2.2 Two Flavors of Private Pointers

We designed and implemented the private pointers API, based on the zero-copy philosophy underlying the access model for private global slots (cf. Sect. 3.4.3). According to this model, a private address is logically indistinguishable from its corresponding stored pointer, therefore a private address and its stored pointer should be regarded as different references to the same slot. As a quick demonstration of this concept, no mechanism is provided by the GAM API to store the content of a local memory value v to a private slot γ, unless v is referred by the stored pointer for γ. We refer to a private address and the corresponding stored pointer as, respectively, global and local references to the same memory slot. Therefore, if a private pointer exists referencing a private slot, in order to ensure that it retains *exclusive* control over the slot, we should prevent the coexistence with *any* other reference, either global or local.

To this aim, we represent private pointers by means of two types, representing respectively global and local references. A global private pointer is represented by an object of type `private_ptr<T>`, where the template parameter `T` represents the type of the referenced memory content. Global private pointers are analogous to public pointers in that they wrap and emulate global pointers. Instead, a local private pointer is represented by an object of the following type:

```
template<typename T>
using gam_unique_ptr = std::unique_ptr<T, void(*)(T *)>;
```

The definition shows how we just exploited the uniqueness semantics of C++ unique pointers for implementing local private pointers. Namely, we defined local private pointers as synonyms for C++ unique pointers with custom deleter. Indeed, as we discuss in Sect. 4.2.3, we embed the release logic in the deleter attached to each local private pointer.

In this setting, we introduce the following *control uniqueness* invariant: given a private slot mapped at global address γ, at any time it exists either a `private_ptr` object containing a (global) pointer to γ or a `gam_unique_ptr` object containing the (local) stored pointer for γ.

4.2.3 API

Similarly to public pointers, global private pointers can be created by means of either the constructor, the generator function `make_private` (or an equivalent template constructor, as discussed in Sect. 4.1.2), or the pulling function `pull_private`. In all cases, the private pointer being constructed has exclusive control over the contained slot.

Following the principles underlying C++ unique pointers, we forbid both copy construction and copy assignment of global private pointers:

```
private_ptr(const private_ptr &) = delete;
private_ptr &operator=(const private_ptr &) = delete;
```

Indeed, copying would imply duplication that in turn would break the uniqueness invariant. Note that copying is forbidden also for local private pointers since C++ unique pointers are non-copyable objects.

Private pointers can be move-constructed and move-assigned:

```
private_ptr(private_ptr &&p) noexcept;
private_ptr &operator=(private_ptr &&p) noexcept;
```

Moving a private pointer `p` represents the action of transferring the exclusive control over slot controlled by `p` to another private pointer, which is either the one being constructed (in case of move construction) or the one `p` is being assigned to (in case of move assignment). The effect of a move-based primitive with argument `p` is twofold: first it causes the resulting private pointer to acquire exclusive control over the slot previously controlled by `p`; second it causes `p` to release control over the slot. Again, such move-based primitives are implicitly provided by unique pointers for local private pointers.

As we anticipated in Sect. 4.2.1, transferring control per se has no effect on memory releasing. Instead, the implicit overwriting induced by assignment causes the control over the formerly controlled slot to be definitively dropped and, hence, it triggers the un-mapping of the slot. Therefore, we identify a first dropping mechanism, that releases control over a slot without un-mapping it. Along the lines of shared pointers, we also provide the corresponding `release` primitive, for global private pointers. Releasing simply invalidates the value stored as contained pointer. Most of the functions moving an argument pointer `p` (e.g., move constructor and assignment) call `p.release()` to exonerate it from the responsibility over the slot. Also, pushing a (global) private pointer relies on releasing, since pushing is logically equivalent to a deferred control transfer.

The other dropping mechanism, that also un-maps the slot, is provided by the `reset` function, which resets the pointer after destructing it. For instance, assigning a private pointer `q` to a variable `p` implies resetting of `p`, since overwriting leads the controlled slot to become definitively unreachable. We remark that both `release` and `reset` functions for global private pointers have the same semantics as their counterparts for unique pointers.

When a private pointer, either global or private, is destructed, it triggers the un-mapping of the controlled slot. Depending on whether the destruction is performed by the author executor or not, the un-mapping is performed locally or remotely, according to the distributed protocol for memory releasing we introduced in Sect. 4.2.1.

Finally, we provide conversion of private pointers, from global to local and vice versa. Note that, to fulfill the uniqueness invariant, conversion

leads control over the controlled slot to be transferred from the pointer being converted to the resulting one. Therefore, the pointer being converted is released as part of both directions of conversion. The former direction is provided by the following function:

```
template<typename T>
gam_unique_ptr<T> local();
```

When `local` is called on a (global) private pointer controlling a global memory address γ, it returns a unique pointer whose contained pointer is the stored pointer for γ. This implies that, after the completion, the issuing executor is the author of γ. Therefore, if `local` is called by a non-author executor, it preliminarily performs both a remote load (cf. Sect. 4.1.2) and a remote un-mapping to *withdraw* both the slot and its authorship from its (former) author. We remark that, since local private pointers are regular C++ unique pointers, the value returned by `local` can be processed by plain C++ code, resulting in implicit loads and stores to the global memory slot referenced by the pointer.

As part of the runtime support for private pointers, the author of each private address γ tracks the *bidirectional* mapping between γ and its stored pointer. For instance, let us consider the `return` statement in the current implementation of `local`:

```
auto deleter = [](T *lp) {ctx.unmap(ctx.parent(lp));};
return gam_unique_ptr<T>(lp, deleter);
```

If the local private pointer is destructed, this should trigger the un-mapping of the corresponding slot, identified by its global address. Therefore the `deleter` callback must be able to track a global address back from its stored pointer (`lp` in the listing), as provided by the context function `parent`.

The inverse conversion, namely from local to global private pointers, is provided by the constructor:

```
template<typename T>
private_ptr(gam_unique_ptr<T> &&lp);
```

In addition to releasing the pointer being converted, this simply returns the global pointer for which `lp` is the stored pointer, provided by the context function `parent`.

Table 4.2 summarizes the most relevant functions provided by the API for global private pointers. For each function, the table reports both the corresponding runtime function that is executed under the hood and its effect with respect to distributed memory releasing. As for local pointers, the API is implicitly provided by C++ unique pointers and is actually a subset of the global API where each function has the same semantics as its global counterpart.

4.3 Smartness

In addition to raising the level of abstraction for the interaction between programs and memory, C++ smart pointers also provide some *smartness* in terms of proper memory management, as we recall in Sect. 2.4.1. In particular, they target two common problematic memory issues—memory leaks and dangling pointers—by means of a generic, statically typed interface, that yield programs that are, by construction, free from those issues.

Function	Back-end GAM primitive	Memory releasing
constructor	`mmap_global`	-
move cons.	-	-
`make_private`	`mmap_global`	-
`pull_private`	`pull`	-
destructor	`unmap`[a]	release
move assignment	`unmap`[a]	release
reset	`unmap`[a]	release
release	-	-
`local`	`local`, `unmap`[b]	release[b]
from-local cons.	`parent`	-
`push`	`push`	-

[a] Remotely if issued by non-author executor.
[b] Remotely and only if issued by non-author executor.

TABLE 4.2: Global private pointers API.

In the same vein, smart GAM pointers provide smartness in terms of GAM memory management. As C++ smart pointers facilitate intentional programming by protecting well formed C++ programs against memory errors, so do smart GAM pointers. Nevertheless, smart pointers—either C++ or GAM—do not target programs that deliberately try to break smartness mechanisms. For instance, smart pointers do not support address aliasing, therefore the behavior of a program is undefined in case of aliasing.

In this section we discuss smartness as provided by smart GAM pointers, along two dimensions of possible GAM memory errors: memory leaks (Sect. 4.3.1) and dangling pointers (Sect. 4.3.2). For each dimension, we also show how smartness could be intentionally broken.

4.3.1 Memory Leaks

Memory leak is a type of memory error that occurs when a program incorrectly manages dynamic memory allocation in such a way that memory which is no longer needed is not released. We define GAM memory leaks by mapping the previous definition to the GAM context, as follows.

Definition 3. *A GAM program contains a* (GAM) memory leak *if at some point in the program execution there exists a memory slot mapped at address γ such that none of the executors holds a reference to γ.*

A GAM program is leak-free if it does not contain any leak. In the following, we informally prove that smart GAM pointers guarantee leak-freeness for all memory slots that are accessed exclusively through smart pointers.

In Sect. 4.1.1, we introduced distributed reference counting for public pointers. The proposed protocol maintains a global invariant stating that, at any time, the author of a public address γ knows the number of references to γ among all the executors. On top of such protocol, we implemented public pointers in such a way that the memory slot for γ is un-mapped as soon as the associated counter reaches zero (cf. Sect. 4.1.2). Therefore, public pointers guarantee leak-freeness by binding the life cycle of each public slot to the associated reference counter.

As for private pointers, we based their implementation on a uniqueness invariant (cf. Sect. 4.2.2), stating basically that, at any time, exactly one private pointer (either global or local) can reference a private slot. When such a reference is definitively dropped (e.g., destructing the pointer without passing the reference to another pointer) the un-mapping of the slot is triggered. Therefore, also private pointers guarantee leak-freeness, by binding the life cycle of each private slot to the associated object that controls it uniquely.

Nevertheless, as smartness in terms of leak-freeness can be broken in C++, so it can be done in GAM by misusing smart GAM pointers. For instance, constructing a private pointer `p` by calling the generator function `make_private` and then calling `p.release()` immediately leads to a memory leak since the unique reference for the controlled slot gets irreversibly dropped.

4.3.2 Dangling Pointers

Dangling pointers are commonly defined as pointers that do not point to a valid destination, where the notions of validity and pointer are intentionally left as generic as possible. For instance, in C++ context, a pointer (either raw or smart) to a piece of memory becomes dangling if the memory referenced by the pointer gets freed, typically through another pointer. Here, the destination pointed by `p` is invalid in that the referenced memory should not be accessed in terms of load and store operations through `p`. For instance, the freed memory could have been reclaimed by the allocator and recycled for storing something that has nothing to do with the previous content.

In GAM context, we give validity a precise meaning in terms of access level (cf. definition 1), as follows.

Definition 4. *Given a GAM system state s, a GAM memory slot at address γ is* valid *in s for a smart GAM pointer if the pointer type (i.e., public or private) matches the access level for γ in s.*

For instance, a global slot mapped at the public address γ is valid for a public pointer referencing γ. Conversely, a smart GAM pointer to γ is dangling if the slot at γ is not valid for the pointer. For instance, a slot at γ is not valid for a private pointer referencing γ if either γ is unmapped or public.

We formalize dangling-free program executions as follows.

Definition 5. *An execution θ of a GAM program is* dangling-free *if and only if, in any state $s \in \theta$, there are no dangling pointers in s.*

A program is dangling-free if all its valid executions are dangling-free.

Although we omit exhaustive proofs for the sake of simplicity, we provide a proof sketch that allows to prove dangling-freeness in a direct manner. Namely, a proof of this property would proceed by contraposition and would prove the following:

$$\begin{array}{lcl} \alpha(s,\gamma) \neq \texttt{public} & \Rightarrow & \nexists \text{ public pointer referencing } \gamma \text{ in } s \\ \alpha(s,\gamma) \neq \texttt{private} & \Rightarrow & \nexists \text{ private pointer referencing } \gamma \text{ in } s \end{array} \tag{4.1}$$

Then the proof would proceed by induction on the length of θ. For the inductive step, we would consider all the semantic rules from figure 3.1

yielding such a state s conforming to the premise in 4.1. In the following, we informally prove dangling-freeness by assuming the outlined structure.

Let us consider public pointers first. The only operation switching the capability for an address γ from public to non-public (namely $\perp$) is the un-mapping. For proving the conclusion in 4.1, it is sufficient to prove that γ is un-mapped only when no references for γ exist among the executors, other than the public pointer that triggers the un-mapping. But this holds trivially, since the un-mapping is triggered only when the reference count for γ (provided by distributed reference counting) reaches zero (cf. Sect. 4.1.2).

As for private pointers, in addition to un-mapping, also pushing and publishing operations switch the capability over γ to non-private for the issuing executor. Namely, both un-mapping and pushing switch it to $\perp$, whereas publishing switch it to public. We recall, from the public pointers API in Sect. 4.1.2, that publishing is induced by constructing a public pointer from a private one. In order to prove that no references for γ exist among the executors, it is sufficient to observe that:

- All of un-mapping, pushing and publishing make the involved private pointer drop control over γ;
- The uniqueness invariant for private pointers (cf. Sect. 4.2.2) guarantees that no references for γ exist among the executors, other than the involved private pointer.

We remark that, with respect to guaranteeing dangling-freeness, the condition provided by the uniqueness invariant is sufficient but not necessary. For instance, one could drop uniqueness internally to each executor by implementing local private pointers as C++ shared pointers and test for uniqueness upon pushing and publishing. However, such an approach would result in a more complex API since pushing and publishing would possibly fail in case they are called on non-unique private pointers, hence forcing the calling context to manage failures.

Summary

In this chapter, we presented smart GAM pointers, a C++ API and implementation, in terms of template classes, providing leak-free and dangling-free dynamic allocation of GAM memory objects.

Chapter 5

Parallel Programming with GAM Nets

In this chapter, we introduce *GAM nets*, a parallel programming model based on GAM (cf. Ch. 3) and smart GAM pointers (cf. Ch. 4).

We build upon the well-assessed FastFlow approach (cf. Sect. 2.4.2) by focusing on *stream parallelism* as the baseline for expressing a broad range of parallelism types. A GAM net is a graph of interconnected *processors* and *communicators*, that represent, respectively, *active* entities that perform computations over data and *passive* entities that deliver data among processors. We pursue "performance by design" by letting processors and communicators exchange *capabilities* (i.e., GAM pointers) rather than data.

We materialize GAM nets in a C++ library. Although we designed the implementation on the same line as FastFlow, we opted for adhering more strictly to modern C++ principles, for instance, by avoiding inheritance in favor of type safe generic programming.

This chapter proceeds as follows. In Sect. 5.1, we introduce the abstract GAM nets model. In Sect. 5.2, we present a C++ object-oriented API and implementation of GAM nets. Finally, In Sect. 5.3, we define a series of nets representing common computation patterns.

5.1 GAM Nets

A GAM net is a directed bipartite graph, in which the nodes represent either *processors* (*active* entities) or *communicators* (*passive* entities) and the directed arcs describe communication links—either from processors to communicators or from communicators to processors.

Formally, let P and C be two disjoint sets of processors and communicators, respectively, and L be a set of links:

$$L \subset (P \times C) \cup (C \times P)$$

Then a GAM net is a triple $N = (P, C, L)$, with further constraints described below.

The two directions of a communication, as performed by a processor, represent pushing/pulling a capability over a GAM slot to/from a peer processor, respectively, via the intermediate communicator. However, differently from the basic API presented in Sect. 3.4.3 and the smart pointers API presented in Sects. 4.1.2 and 4.2.3, a processor does not specify its communication peers when pushing or pulling. Rather, a processor specifies

which communicator is involved, among those it is connected to. In this setting, the logic for routing capabilities is embedded in the communicators.

In the context of theoretical models for representing parallel computations, GAM nets are based on the same *channel-centric* view underlying two classical models, namely Calculus of Communicating Systems (CCS) [104] and Communicating Sequential Processes (CSP) [85] models. In CCS and CSP, unnamed active processes communicate through named passive channels, according to a fixed topology. Similarly, in GAM nets, the topology L is fixed and the processors refer to communicators rather than peer processors when communicating.

CSP and CCS were designed to model *asynchronous processes*, i.e., processes that proceed independently at their own speed, with no global clock. However, as discussed by Brooke [41], processes were assumed to interact by means of *synchronous communication*: a process attempting to perform an input action synchronizes with another process attempting a matching output, waiting if necessary until a match becomes available—and vice versa. Therefore, we refer to Brooke's alternative CSP model [41], directly based on *asynchronous* communication channels, as the theoretical basis for GAM nets.

We proceed by defining the components of GAM nets syntax, namely communicators (Sect. 5.1.1) and processors (Sect. 5.1.2). In Sect. 5.1.3, we introduce a dataflow-like execution model for GAM nets.

5.1.1 Communicators

In a GAM net, communicator nodes represent passive links between *input* and *output* processors. As illustrated in Fig. 5.1, the communication between any pair of processors is mediated by an intermediate communicator.

Formally, given a net $N = (P, C, L)$ and a communicator $c \in C$, the sets of input and output processes for c, that we refer to as $\text{in}(c)$ and $\text{out}(c)$, respectively, are defined as follows:

$$\text{in}(c) = \{p \in P : (p, c) \in L\}$$
$$\text{out}(c) = \{p \in P : (c, p) \in L\}$$

We exclude from valid nets the case of a process attached to both input and output side of a communicator:

$$\forall\, c \in C, (p \in \text{in}(c) \Rightarrow p \notin \text{out}(c)) \wedge (p \in \text{out}(c) \Rightarrow p \notin \text{in}(c))$$

This implies that the sets of input and output processors are disjoint:

$$\forall\, c \in C, \text{in}(c) \cap \text{out}(c) = \varnothing$$

We also exclude communicators with no processes attached to either input or output side:

$$\forall\, c \in C, \text{in}(c) \neq \varnothing \wedge \text{out}(c) \neq \varnothing$$

As "passive" entities, the sole role of communicators is to embed some policies to deliver capabilities from input to output processors. Such *dispatching policies* are implicitly invoked by any processor whenever it calls

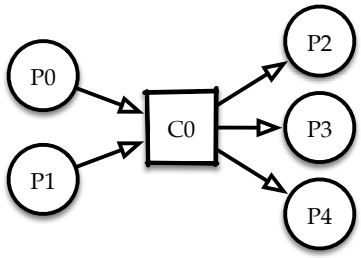


FIGURE 5.1: A communicator C0 in a GAM net, with input processors P0, P1 and output processors P2, P3, P4.

a communication primitive to perform a communication. Instead of giving precise constraints about the form in which policies are expressed, we leave such aspect to the implementation level. Let us consider, as a running example, a hypothetical purely functional implementation of GAM nets.

Pushing Policies

In a purely functional setting, a policy for pushing a (public) address would be a function with the following arguments: the address to be pushed, the input processor, and the sets of output processors. This function would return a set of output processors—a subset of the possible output processors (third argument)—to which the address will be delivered. Thus, its signature would be:

$$f : \Gamma \times P \times 2^P \to 2^P \tag{5.1}$$

This function would then be invoked, upon pushing address γ via communicator c by processor $p \in \text{in(c)}$, as follows:

$$f(\gamma, p, \text{out(c)}) = o \tag{5.2}$$

Each of the output processors $p' \in o \subseteq \text{out(c)}$ identified by the function would then be used by the runtime as a destination for delivering the capability over γ. We refer to the f in the above example as a *multicast* policy, since the capability is possibly delivered to multiple destinations.

A common multicast policy, generally referred to as *broadcast*, delivers each capability to all the output processors:

$$f(\gamma, p, \text{out}(c)) = \text{out}(c) \tag{5.3}$$

In a simpler class of pushing policies, that we refer to as *unicast* or *switch* policies, the capability is delivered to a single destination:

$$f(\gamma, p, \text{out(c)}) = o \;\; \text{where} \; |o| = 1 \tag{5.4}$$

Obviously, only unicast policies make sense for pushing private addresses, since multicast delivery implies replication of the capability at hand, which is not allowed for private addresses.

The most basic unicast policy is the constant function, that delivers all capabilities to the same output processor $p' \in \text{out}(c)$. In particular, the constant function is the only viable option for such communicators c attached to a single output processor, so that $|\text{out}(c)| = 1$.

More complex policies can be defined that take as input additional parameters for dispatching. A common example is the *key-partitioning* policy, that takes as input a key and delivers to the same processor all the capabilities associated to the given key, according to some dispatching function g. For instance, keys could be mapped to output processors in round-robin fashion, resulting in the following dispatching policy, where K is the set of keys and where we assume processors in $out(c)$ are numbered from 0 to $|out(c)| - 1$:

$$\begin{array}{rccc} f: & \Gamma \times P \times 2^P \times K & \rightarrow & 2^P \\ & f(\gamma, p, \text{out}(c), k) & = & \{p_{k \bmod |out(c)|}\} \end{array} \tag{5.5}$$

Pulling Policies

As for pulling, the policy determines which processor to pull from:

$$g : P \times 2^P \rightarrow P \tag{5.6}$$

This function would be invoked, upon pulling via communicator c by processor $p \in \text{out(c)}$, as follows:

$$g(p, \text{in(c)}) = p' \tag{5.7}$$

The input processor $p' \in \text{in(c)}$ would then be used by the runtime as the source for pulling a capability.

On the same line as unicast pushing policies, the most basic pulling policy is the constant function, in which all capabilities are pulled from the same input processor $p' \in \text{in}(c)$. In particular, the constant function is the only viable option for such communicators c attached to a single input processor, such that $\text{in}(c) = \{p'\}$.

Dually, the *nondeterminate merge* (as defined by Lee *et al.* [96]) is a nondeterministic policy in which any input processor may be selected as the one to pull from.

5.1.2 Processors

In addition to communicator nodes, a GAM network is composed of processor nodes, that represent active processing entities.

As illustrated in Fig. 5.2, processors are attached to a global memory that they possibly access according to the GAM model. During its lifetime, each processor performs arbitrary computations, that include exchanging capabilities with peer processors via intermediate communicators. For the sake of simplicity, we reduce the set of valid nets by imposing that any processor is attached to at least one communicator, thus we exclude isolated processors:

$$\forall p \in P, \exists c \in C : (c, p) \in L \vee (p, c) \in L$$

Moreover, we do not allow processors that are attached to two different communicators on the same side:

$$\forall p \in P, (\exists c \in C : (c, p) \in L) \Rightarrow (\nexists c' \in C : (c', p) \in L)$$
$$\forall p \in P, (\exists c \in C : (p, c) \in L) \Rightarrow (\nexists c' \in C : (p, c') \in L)$$

In this setting, we categorize processors in terms of their role with respect to communication:

- *Source* and *sink* processors may only push/pull capabilities to/from the communicator to which they are attached, respectively;
- *Filter* processors may communicate with the two communicators they are attached to, either pulling from one communicator or pushing to the other.

This categorization can be formalized in terms of attached communicators as follows. Given a net $N = (P, C, L)$ and a processor $p \in P$, exactly one of the following conditions holds:

$$\begin{array}{lcl} p \text{ is a source} & \Leftrightarrow & \exists c \in C : (p, c) \in L \wedge \nexists c' \in C : (c', p) \in L \\ p \text{ is a filter} & \Leftrightarrow & \exists c, c' \in C : (c, p) \in L \wedge (p, c') \in L \\ p \text{ is a sink} & \Leftrightarrow & \nexists c \in C : (p, c) \in L \wedge \exists c' \in C : (c', p) \in L \end{array}$$

For brevity, we refer to a communicator c as *input communicator* for p if $(c, p) \in L$ and, conversely, as *output communicator* for p if $(p, c) \in L$. In Fig. 5.2, P0 is a source since it is attached only to output communicator C0; similarly, P2 is a sink since it is attached only to input communicator C1; finally, P1 is a filter since it is attached to both input communicator C0 and output communicator C1.

5.1.3 Execution Model

We recall that, according to the asynchronous CSP model [41] that we consider as reference, processors execute independently and exchange capabilities by means of asynchronous communications.

This allows to naturally define a *dataflow-like* execution model for GAM nets, in which each processor is characterized by a *kernel* that either is activated each time a capability is delivered to the processor (filters and sinks) or internally generates capabilities (sources). Within the execution of a kernel, one or more capabilities can be emitted to downstream processors, via the output communicator of the processor. In this setting, the coordination of processors is driven by the flow of data—representing data dependencies—rather than through explicit synchronizations.

From the perspective of the proposed dataflow-like execution model, communicators define the *activation policy* for the respective output processors. For instance, let us consider a communicator c with key-partitioning pushing policy (5.5) and nondeterminate merge pulling policy. This communicator, that we refer to as *shuffle*, leads the kernel of each output processor p to be activated for each capability γ, emitted with key k by some input processor p_i, such that:

$$f(\gamma, p_i, \text{out}(p), k) = \{p\}$$

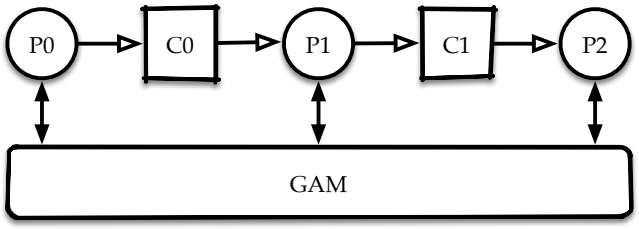


FIGURE 5.2: A GAM net composed of processors P0, P1, P2 attached to a GAM memory, communicating pairwise via intermediate communicators C0, C1.

The activation policy also defines the *order* in which capabilities are processed by the output processors. For instance, the shuffle communicator guarantees that, given an output processor p, the capabilities emitted by an input processor p_i are presented to p in the order in which they were emitted. Note that the mentioned order, as seen by a given output processor p, is partial in case of multiple input processors, whereas it is total in case of single input processor.

5.2 C++ Implementation

In this section, we present a C++ API for GAM nets, together with its implementation.

Targeting zero-overhead abstraction, in fulfillment of modern C++ principles, we designed both the API and the implementation around generic programming, thus avoiding any overhead arising from dynamic polymorphism.

Since both GAM nets and their implementation are based on the same principles as FastFlow (cf. Sect. 2.4.2), we refer to the proposed implementation as GFastFlow (for GAM-FastFlow) and we enclose the associated C++ code in a the `gff` namespace.

We proceed by introducing the GFastFlow API (Sect. 5.2.1), followed by some details about the implementation (Sect. 5.2.2).

5.2.1 API

In Sect. 5.1.3, we introduced a dataflow-like execution model for GAM nets, in which a processor is characterized by its role with respect to communication (i.e., source, filter, or sink) and by a kernel that processes delivered or generated capabilities, one at a time. Moreover, according to the proposed execution model, a communicator describes how the emitted capabilities lead to activating the kernels of output processors.

In the following, we present the API of GFastFlow, a C++ library that implements GAM nets, from the perspective of the proposed execution model.

Processors

GFastFlow processors are objects of either `Source`, `Filter`, or `Sink` template classes, each representing processors with the corresponding role:

```
template<typename OutComm,
         typename out_t,
         typename ProcessorLogic>
class Source;

template<typename InComm, typename OutComm,
         typename in_t, typename out_t,
         typename ProcessorLogic>
class Filter;

template<typename InComm,
         typename in_t,
         typename ProcessorLogic>
class Sink;
```

The `InComm` and `OutComm` template parameters represent the type of, respectively, the input and output communicator to which the processor is attached. The `in_t` parameter represents the smart pointer type corresponding to the capabilities delivered to the processors (via the input communicator) and processed by the kernel. Conversely, the `out_t` parameter represents the smart pointer type corresponding to the capabilities emitted by the processor (to the output communicator) within each kernel execution. Finally, the `ProcessorLogic` parameter represents the type of the business logic of the processor, that we discuss in the following.

The complete specification of a processor includes its kernel (i.e., the way it processes or generates capabilities) and its *termination* logic. Indeed, according to the process network execution model, that in turn underlies the dataflow model we are considering, processors execute their kernels as long as some termination condition is not met. Within the business logic of a processor, both the kernel and the termination logic are embedded in a single `svc` function, along with `svc_end` for post-termination processing, as we discuss later.

Termination

The termination logic is expressed in terms of special values to be returned by `svc` functions, that we refer to as *termination tokens* in accordance with the token-based dataflow terminology. A termination token is a value of type `token_t` and, in the current API, it can be either `go_on` or `eos`, representing the fact that the processor should keep executing or should terminate, respectively, after the `svc` instance at hand is completed.

Based on those termination tokens, the termination protocol is then as follows: when a `svc` function from a processor `p` returns an `eos` token, that token is propagated to *all* the processors downstream of `p` via the intermediate communicators; furthermore, when a processor receives an `eos` token, its `svc` function is not called anymore. Conversely, a `go_on` token is returned by a `svc` instance to indicate that the respective processor should keep executing.

Communicators

The API provides a set of built-in communicators, each characterized by an activation policy (cf. 5.1.3). For instance, the `OneToOne` communicator is characterized by the simplest activation policy, stating that the `svc` function of the output processor is activated for each capability emitted by the input processor, and the emission order is respected. We remark that the activation policy is a *semantic* characterization of each communicator, defined in terms of the underlying execution model, thus invisible from the API viewpoint.

Communicators are accessed within the `svc` function of source and filter processors, in order to emit capabilities. To this aim, each communicator type provides an `emit` function. In case of emitting processors (i.e., sources and filters), the attached output communicator is passed to the `svc` function as input parameter, so that the `emit` function can be called.

In its simplest form, the `emit` function takes as input only the capability to be emitted. In this case, the signature for private pointers is as follows:

```
template<typename T>
void emit(gam::private_ptr<T> &&);
```

However, some communicators rely on more sophisticated pushing policies, leading to a richer signature for `emit`. For instance, the `emit` function for the `Shuffle` communicator, whose activation policy we described in Sect. 5.1.3, requires a parameter of type `K`, representing the key type. In this case, the signature for private pointers is as follows:

```
template<typename T, typename K>
void emit(gam::private_ptr<T> &&, const K &);
```

The signatures of the `emit` functions provided by a communicator type also determine which types of capabilities can be emitted. In particular, multicast communicators (cf. Sect. 5.1.1) allow to emit only public pointers, therefore no signatures for private pointers must be provided by multicast communicators.

In Sect. 5.3, we provide some examples in order to show the expressiveness of various built-in communicators.

Processor Logic

From the discussion above, the `svc` signatures for source, filter, and sink processors are as follows, where `in_t` is the type of the input capability to be processed (e.g., `gam::private_ptr<int>`) and `OutComm` is the type of the output communicator:

```
gff::token_t svc(OutComm &); //source
gff::token_t svc(in_t &, OutComm &); //filter
void svc(in_t &); //sink
```

Note that the `in_t` parameter is missing in the source case, since sources do not receive capabilities—they generate them. Conversely, both the return and the communicator types are missing in the sink case, since sinks have no downstream processors.

In addition to the `svc` function, that embeds both the kernel and the termination logic of the processor, the specification of the processor logic includes the `svc_init` and `svc_end` functions. The former is called by the

runtime before entering the main loop (i.e., the loop in which `svc` is repeatedly invoked), whereas the latter is called after exiting the main loop.

In order to compose a processor logic, the functions `svc`, `svc_init`, and `svc_end` are packed into a single `ProcessorLogic` object, whose type parametrizes the processor type. Objects representing processor logic are constructed by the runtime by means of their default constructors, therefore they must be default-constructible.

In this setting, we refer to processors of type `Source`, `Filter`, or `Sink` as *stateful processors*, where the state is represented by the data members of the processor logic. In particular, the `svc` function may access the state to implement non-functional behaviors, whereas `svc_init` and `svc_end` functions may perform custom initialization and finalization of the state, respectively.

Nets

For the sake of simplicity, in the current API we only support programs consisting of a single GAM net. Therefore, the API for building nets consists in a single function `add`, which takes as argument the processor object (of parametric template type) to be added. Once all processors have been added, the execution of the net can be triggered by calling the `run` function, that executes the net composed by the processors that have been `add`-ed.

The topology of a net is defined in an incremental manner by passing (references to) communicator objects to processor constructors, whose signatures are as follows:

```
Source(OutComm &);
Filter(InComm &, OutComm &);
Sink(InComm &);
```

In particular, two processors `p` and `q` are attached to the same communicator `c` if a reference to `c` is passed to both constructors for `p` and `q`.

For instance, let us consider the simple topology consisting of a single source processor (of type `P`) attached to a single sink processor (of type `Q`) through a communicator with the constant function as both push and pull policies. Listing 5.1 shows how such a topology is defined and executed. Note that the processor's logic is not described: some examples will be provided in Sect. 5.3; note also that a reference to communicator `c` is passed to the constructors of both source and sink objects.

5.2.2 Implementation

In the following we provide some details about the implementation of the GFastFlow library, whose API we presented in Sect. 5.2.1. In particular, we focus on the modular implementation of communicators, based on template programming, which allows to define custom communicators with limited effort.

As the fundamental abstraction underlying the implementation of GAM nets, processors are implemented as GAM executors (i.e., entities exchanging capabilities), whereas communicators represent sets of logical links connecting executors. Therefore, at implementation level, GAM net processors are identified by their respective executor's identifier.

```
/* The producer processor. */
using P =
gff::Source<gff::OneToOne, // out communicator type
gam::private_ptr<int>,     // out capability type
PLogic>;                   //processor logic

/* The consumer processor. */
using Q =
gff::Sink<gff::OneToOne,   // in communicator type
gam::private_ptr<int>,     // in capability type
QLogic>;                   //processor logic

int main()
{
    /* Construct a one-to-one communicator. */
    gff::OneToOne c;

    /* Build a procuder-consumer net. */
    gff::add(P(c));
    gff::add(Q(c));

    /* Execute the net. */
    gff::run();

    return 0;
}
```

LISTING 5.1: Skeleton of a producer-consumer GAM net.

Communicators

In the proposed implementation, an object implementing a GAM communicator includes some variants of the `emit` function (e.g., for public and private pointers) and an object of the template class `CommunicatorInternals`, whose (partial) signature is shown in Listing 5.2. Objects of this class represent the *communication back end* for the respective enclosing communicator. We remark that, at implementation level, processors are identified by GAM executors, as in line 22.

Note that the primitives provided by `CommunicatorInternals` objects deal with any kind of pointer, thus they support any form of communication between processors. Namely, a pointer can be:

- *pulled* by an output processor from an input processor, via the intermediate communicator, by means of the `get` primitive;
- *pushed* by an input processor to an output processor, via the intermediate communicator, by means of the `put` primitives;
- *pushed* by an input processor to all the output processors by means of the `put_to_all` primitives.

As for the implementation of `emit` functions, it typically consists in a single call to `put`, with the same arguments as the calling `emit`. Different signatures for `emit` are supported by means of the template parameter pack `PolicyArgs`, that allows to forward the additional arguments from the `emit` call to the dispatching policy call, as we discuss later.

```
template<typename PushDispatcher, typename PullDispatcher>
class CommunicatorInternals {
public:
  template<typename T, typename ... PolicyArgs>
  void put(const gam::public_ptr<T> &p, PolicyArgs&&...);

  template<typename T, typename ... PolicyArgs>
  void put(gam::private_ptr<T> &&p, PolicyArgs&&...);

  template<typename T>
  void put_to_all(const gam::public_ptr<T> &p);

  template<typename T>
  void put_to_all(gam::private_ptr<T> &&p);

  template<typename ptr_t>
  ptr_t get();

private:
  PushDispatcher push_dispatcher;
  PullDispatcher pull_dispatcher;
  std::vector<gam::executor_id> input, output;
};
```

LISTING 5.2: Prototype of the `Communicator` template class.

Dispatchers

Internally, back end objects rely on push and pull dispatchers, that are represented by objects of template type `PushDispatcher` and `PullDispatcher`, respectively. Dispatchers embed both the communication logic (i.e., the interaction with the GAM runtime) and the dispatching policy (cf. Sect. 5.1.1). In particular, `put` and `put_to_all` primitives in Listing 5.2 rely on the push dispatcher, while `get` relies on the pull dispatcher.

When invoking a dispatcher primitive, the calling communicator primitive passes as argument the set of either input or output processors, in form of vectors (i.e., `input` and `output` in Listing 5.2) that represent numbering. Again, additional arguments of the calling function (i.e., a back end primitive) are forwarded as parameter pack.

Two different families of push dispatchers are represented by objects of the `Switch` and `Multicast` template classes, embedding switch and multicast policies as defined in (5.4) and (5.2), respectively. Since the duplication of private pointers is not allowed, only public pointers can be exchanged via multicast dispatchers. Therefore, a communicator composed of a multicast dispatcher does not implement the `put` primitive for the private pointer case. Indeed, we refer to such a communicator as a *public-only* communicator.

As for pull dispatchers, they are represented by objects of the `Merge` template class, embedding a pulling policy as defined in (5.6).

Dispatching Policies

Dispatching policies are represented by C++ callable objects, thus they are expressed in a generalized functional style, similar to those introduced in Sect. 5.1.1.

For instance, a simplified[1] implementation of a constant switch policy, always selecting the processor with index 0, is the following lambda:

```
auto ConstantSwitchPolicy =
[] (const vector<gam::executor_id> &d) {
    return d[0];
};
```

Another common policy is the round-robin switch, that can be implemented in a compact way by exploiting the flexibility of callable objects, as shown in the following listing:

```
class RoundRobinSwitchPolicy {
public:
  gam::executor_id operator()(const vector<gam::executor_id> &d) {
    gam::executor_id res = d[rr_cnt];
    rr_cnt = (rr_cnt + 1) % d.size();
    return res;
  }

  gam::executor_id rr_cnt = 0;
};
```

Built-in dispatchers are listed in Table 5.1. We remark that, as can be seen in the table, dispatchers embedding policies that differ only by their input signatures (e.g., `RRTo` and `ByKeyTo`) are represented by the same family, that is, the same template class (e.g., `Switch`). This form of polymorphism is based on C++ variadic templates, that allows to define parametric functions with respect to the input signature (e.g., the `put` functions in Listing 5.2). In particular, input arguments are expressed as *universal references* (denoted by `&&`), that allows *perfect forwarding* (i.e. preserving value categories). For each push communication, (parametric) input arguments are forwarded from the outermost call at processor side to the innermost invocation of the dispatching policy.

Moreover, in the proposed implementation, the output signatures of switch and multicast policies differ in that the former returns a single processor, whereas the latter returns a set of processors, as suggested by the definitions in Sect. 5.1.1. Therefore, a basic policy (i.e., with no additional arguments) for a multicast dispatcher would have the following format, where the second parameter represents the set of output processors selected by the policy:

```
auto CustomMulticastPolicy =
[](const vector<gam::executor_id> &, vector<gam::executor_id> &) {
    /* .... */
};
```

Custom Communicators

The dispatching policy is embedded into dispatcher types as a template parameter. Therefore, defining a custom dispatcher in addition to the built-in ones amounts to defining a policy in the form of a C++ callable, whose output signature complies with the dispatcher family. Since policy objects are built based on their respective default constructor, according to C++

[1] For the sake of simplicity, we implemented dispatching policies with limited signatures with respect to their functional counterparts in Sect. 5.1.1. Namely, we omitted all the parameters but input and output processors from, respectively, signatures (5.1) and (5.6).

Type	Family	Policy Signature	Peer(s) Selection
		Push Dispatchers	
`To`	`Switch`	$2^P \to P$	constant
`RRTo`	`Switch`	$2^P \to P$	round robin
`Broadcast`	`Multicast`	$2^P \to 2^P$	all
`ByKeyTo`	`Switch`	$2^P \times K \to P$	key-based hashing
`RRMulti`	`Multicast`	$2^P \to 2^P$	group-wise round robin
		Pull Dispatchers	
`From`	`Merge`	$2^P \to P$	constant
`RRFrom`	`Merge`	$2^P \to P$	round robin
`FromAny`	`Merge`	$2^P \to P$	any

TABLE 5.1: Built-in dispatchers. For each dispatcher, the table shows, from left to right, the C++ type, the family of dispatchers it belongs to, the logical signature of its dispatching policy, and a brief description of its behavior.

they must be `DefaultConstructible`, that is, they must provide a default constructor. Note that future C++ versions will allow to directly associate this kind of compile-time constraints to template classes (and functions), by means of the *concepts* language feature.

Given a custom policy of type `policy`, with its specific input signature, defining a custom communicator is straightforward and amounts to assembling a class including, as public member named `internals`, a back end object of template class `CommunicatorInternals`, parametrized with the desired pushing and pulling dispatchers. In particular, the type of the pushing dispatcher should be a family (e.g., `Switch`) parametrized with `policy`, such that the output signature identified by the family complies with the output signature of `policy`. Finally, some variants of `emit` should be provided, with signatures that match the input signature(s) of callable `policy` objects. As an example, Listing 5.3 shows the definition of a custom variant of the `Shuffle` communicator, where the mapping from keys to output processors is encoded by a custom hashing function `h`.

Processors

The implementation of a GAM processor consists mainly in a *routine*, called `run` in the current implementation, that realizes the required dataflow-like behavior for each type of processor—i.e., `Source`, `Filter`, and `Sink`.

Listing 5.4 shows the `run` routine for `Filter` processors. The first and last instructions (lines 2 and 30) call the initialization and finalization functions `svc_init` and `svc_end`, respectively. As discussed in Sect 5.2.1, both functions are provided by the `ProcessorLogic` type, that represents the business logic of the processor. Within the object implementing the processor, an instance of the business logic is constructed and stored as the member object named `logic`.

Then the routine enters a loop that repeats the following steps until termination is detected by receiving an appropriate `eos` token:

1. Obtain a capability by pulling it from the input communicator (line 9);

```
/* Custom hash-based dispatching policy. */
auto HashPolicy =
[](const vector<gam::executor_id> &d, const K &key) {
   return h(key) % d.size();
};

class CustomShuffle {
public:
    /* emit function for private pointers */
    template<typename T, typename K>
    void emit(gam::private_ptr<T> &&p, const K &key) {
        internals.put(std::move(p), key);
    }

    /* emit function for public pointers */
    void emit(const gam::public_ptr<T> &p, const K &key) {
        internals.put(p, key);
    }

    /* Communication back end object */
    CommunicatorInternals<Switch<HashPolicy>, FromAny> internals;
};
```

LISTING 5.3: Custom variant of the `Shuffle` communicator.

2. Call the `svc` function (i.e., the kernel) with the obtained capability as argument (line 14);

We remark that, since communicator objects only provide `emit` functions, pulling of capabilities is performed exclusively by the runtime. Conversely, pushing has to be explicitly invoked within the `svc` function and no implicit pushing is performed by the runtime.

The remaining code in `run` is dedicated to handling termination, according to the protocol depicted in Sect. 5.2.1. Exiting the indeterminate loop occurs either because the `svc` function returns an `eos` token (line 17) or because all the upstream processors have terminated (line 26). To this aim, a termination token is *propagated* by broadcasting it to all downstream processors, as non-mappable GAM values[2] (cf. Sect. 3.4.3), upon exiting the loop (line 33).

5.3 Net Patterns

In this section, we show how GAM nets can be exploited to implement two well-known patterns in the context of structured parallel programming, namely, *pipeline* (Sect. 5.3.1) and *farm* (Sect. 5.3.2). On top of the presented patterns, we introduce *active communicators* (Sect. 5.3.3) as a mechanism for providing scalable communication among GAM processors.

[2]Non-mappable GAM values do not represent references to memory locations, therefore they are not subject to memory rules. For instance, a non-mappable value wrapped into a private pointer can be passed to multiple processors, as with broadcasting `eos` tokens.

```
void run() {
    logic.svc_init(); //prelude user code

    in_t in;
    token_t out;

    while (true) {
        /* Pull a capability via input communicator. */
        in = in_comm.internals.template get<in_t>();

        if (!is_eos(in)) { //meaningful input capability

            /* Process the capability by invoking the kernel. */
            out = logic.svc(in, out_comm);

            if (is_eos(out)) //flagged termination
                break;

            //kernel returned go_on: continue

        } else { //got eos from input communicator
            ++got_eos;

            /* Check if all upstream processors terminated. */
            if (got_eos == in_comm.internals.in_cardinality())
                break;
        }
    }

    logic.svc_end(); //postlude user code

    /* Propagate eos token. */
    out_comm.internals.broadcast(make_eos<out_t>());
}
```

LISTING 5.4: Routine of `Filter` processors.

5.3.1 Pipeline

Among the common forms of parallelism, pipelining is a method for parallelizing sequential computations by segmenting them into a series of sequential *stages*. Parallelism is achieved by running each stage simultaneously on consecutive data elements, that are processed in the same order as the one in which they enter the pipeline.

Mapping a pipeline to a GAM net is straightforward, since it amounts to mapping each stage of the pipeline to a GAM processor and attaching the processors pairwise by means of one-to-one communicators (i.e., `OneToOne` in Listing 5.1). As for the type of the processors, all the internal stages are mapped to filters, whereas the first and the last stage are mapped to a source and a sink, respectively.

Figure 5.3 illustrates a four stages pipeline in which the business logic for each stage is as follows:

1. Generate a random stream of numeric values, all within a given range;
2. Filter out the values above a given threshold;
3. Compute the square root;
4. Check the result.

Listing 5.5 shows the code for the `RandomStreamGenerator` class, the first stage of the pipeline (RSG in Fig. 5.3). Since it is a source processor, the `svc` function within the business logic class is invoked repeatedly by the runtime, until it signals termination by returning an `eos` token. For each activation, until the stream has been completely generated and emitted (i.e., unless condition at line 9 holds), a random integer value is generated and packed into a private pointer (line 13); then the pointer is emitted via the one-to-one communicator (line 13) and a `go_on` token is returned to indicate the computation should proceed (line 14). Note that the private pointer is *moved* upon passing it to the `emit` function.

Listing 5.6 shows the code for the `Lowpass` class, the second stage of the pipeline (LP in Fig. 5.3). Since it is a filter processor, the `svc` function within the business logic class is invoked by the runtime each time a private pointer is delivered to the corresponding processor by the upstream communicator. The `svc` function takes as input the pointer to be processed and converts the pointer from global to local form (line 5), to turn into a regular shared-memory pointer; then, if the value carried by the pointer is below a given threshold (line 6), the pointer is casted back to global form, by means of the `private_ptr` constructor, and emitted via the second communicator (line 7); finally, a `go_on` token is returned to indicate the computation should proceed to process the next delivered pointer.

Note that, in class `LowpassLogic`, when the control reaches the end of a `svc` instance, the private pointer in its local form (i.e., the `local_in` object) is destructed by calling the `std::unique_ptr` destructor. At this point, the destruction has no effect if the pointer has been previously casted to global form (line 5), since the conversion also releases control over the controlled slot, according to the move semantics we defined for private pointers. On the other hand, if the destructor is called without the conversion having occurred (i.e., the condition at line 6 did not hold), then the destructor causes

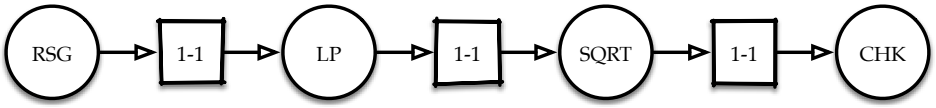


FIGURE 5.3: GAM net representing a 4-stage pipeline.

the corresponding slot to be unmapped remotely (cf. Sect. 4.2.3) and the corresponding memory (allocated from the local memory of the upstream processor) to be freed.

Listing 5.7 shows the code for the `SquareRoot` class, the third stage of the pipeline (SQRT in Fig. 5.3). This processor logic implements a straightforward functional computation, sometimes referred to as *apply-to-all*, in which each value delivered to the processor is accessed (by first converting the incoming pointer to local form), then a function is computed on the value, and finally the result is emitted downstream (after being converted back to global form). Note that, since no releasing is performed on the incoming pointer after it has been converted to local form, the slots controlled by each incoming pointer are un-mapped (thus causing the corresponding allocated memory to be freed) remotely when the control reaches the end of the `svc` function.

Finally, Listing 5.8 shows the code for the `Checksum` class, the fourth and last stage of the pipeline (CHK in Fig. 5.3). The `svc` function within the business logic for sink processors is invoked by the runtime according to the same criterion as for filter processors, that is, each time a pointer is delivered to the processor. In this case, the `svc` accesses the value carried by the incoming pointer and sums it up into a local accumulator, that the processor logic stores as part of its local state (i.e., the `sum` member). This logic also shows how the `svc_end` function can be exploited to perform some post-processing, which in this case consists in computing the expected value for the accumulator (by replicating locally the entire computation performed by the pipeline) and comparing it to the final value of the `sum` member. Also in this case, the slots controlled by each incoming pointer are un-mapped at the end of the each `svc` instance.

In Sect. 7.1 we show how GAM pipelines allow to express a variety of parallel applications.

5.3.2 Farm

Another common form of parallelism arises from executing multiple independent tasks in parallel, for instance, filtering multiple items from a stream in parallel, using functional replication. This form of parallelism is captured by the *farm* skeleton, which consists of a *master* and a pool of *workers*—farm is also known as *master–workers*. The master is responsible for distributing the input tasks towards the worker pool, as well as for gathering the partial results to produce the final result of the computation. A worker entity gets an input task, processes the task, and sends the result back to the master. Moreover, in order to introduce pipelining between the distribution activity and the gathering activity, the master is in turn split into two entities, namely, the *scheduler* and the *collector*.

```
class RandomStreamGeneratorLogic {
private:
    unsigned n = 0;
    std::mt19937 rng; // Mersenne Twister pseudo-random generator
    std::uniform_int_distribution<int> d {0, LIMIT};

public:
    gff::token_t svc(gff::Emitter<OneToOne> &e) {
        if (n == STREAMLEN) //check for termination
            return gff::eos;

        ++n;
        e.emit(gam::make_private<int>(d(rng)));
        return gff::go_on;
    }

    void svc_init() {}
    void svc_end() {}
};

typedef gff::Source<gff::OneToOne,
        gam::private_ptr<int>,
        RandomStreamGeneratorLogic> RandomStreamGenerator;
```

LISTING 5.5: Random stream generator for the GAM pipeline example.

```
class LowpassLogic {
public:
    gff::token_t svc(gam::private_ptr<int> &in,
                     Emitter<OneToOne> &e) {
        auto local_in = in.local();
        if (*local_in < THRESHOLD)
            e.emit(gam::private_ptr<int>(std::move(local_in)));
        return gff::go_on;
    }

    void svc_init() {}
    void svc_end() {}
};

typedef gff::Filter<gff::OneToOne, gff::OneToOne,
        gam::private_ptr<int>, gam::private_ptr<int>,
        LowpassLogic> Lowpass;
```

LISTING 5.6: Lowpass filter for the GAM pipeline example.

```
class SquareRootLogic {
public:
    gff::token_t svc(gam::private_ptr<int> &in,
                     Emitter<OneToOne> &e) {
        float res = std::sqrt(*(in.local()));
        e.emit(gam::make_private<float>(res));
        return gff::go_on;
    }

    void svc_init() {}
    void svc_end() {}
};

typedef gff::Filter<gff::OneToOne, gff::OneToOne,
        gam::private_ptr<int>, gam::private_ptr<float>,
        SquareRootLogic> SquareRoot;
```

LISTING 5.7: Square root calculator for the GAM pipeline example.

```
class ChecksumLogic {
private:
    float sum = 0;
    std::mt19937 rng; //same random sequence as stream generator

public:
    void svc(gam::private_ptr<float> &in) {
        sum += *(in.local());
    }

    void svc_init() {}

    void svc_end() {
        float res = 0;
        for (unsigned i = 0; i < STREAMLEN; ++i) {
            int x = (int) (rng() % RNG_LIMIT);
            if (x < THRESHOLD)
                res += sqrt(x);
        }
        assert(res == sum);
    }
};

typedef gff::Sink<gff::OneToOne,
        gam::private_ptr<float>,
        ChecksumLogic> Checksum;
```

LISTING 5.8: Result checker for the GAM pipeline example.

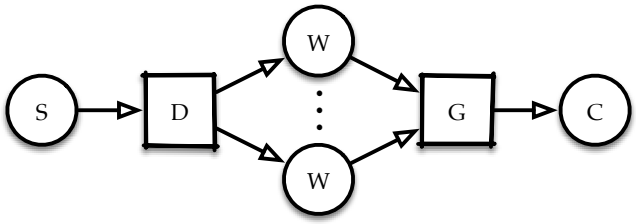


FIGURE 5.4: A GAM net representing a farm.

Fig. 5.4 illustrates a GAM net implementing a farm. Each entity of the farm skeleton is mapped to a GAM processor as follows (letters between parentheses refer to identifiers in Fig 5.4):

- The farm scheduler is mapped to a source processor (S);
- Each farm worker is mapped to a filter processor (W);
- The farm collector is mapped to a sink processor (C).

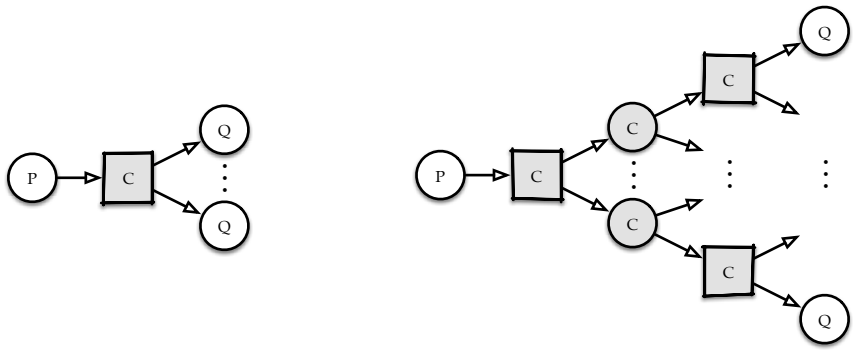
The scheduler S is linked to each worker by an one-to-many communicator (D), whereas each worker is linked to the collector C by a many-to-one communicator (G). Since D is attached to a single input processor, the pulling policy for D is the constant function that always returns S, as discussed in Sect. 5.1.1. Symmetrically, the pushing policy for G is the constant function that always returns C.

As for pushing and pulling policies for, respectively, communicators D and G, they depend on the specific type of farm being implemented, as we discuss in Sect. 7.1.

5.3.3 Active Communicators

Let us consider the scenario in which a processor p is linked to a number of downstream processors by an one-to-many communicator. Let us also assume that the pushing dispatcher is of `broadcast` type (cf. Sect. 5.2.2), that is, each incoming pointer is delivered to all the output processors. We remark that the depicted scenario is not an unrealistic one. For instance, in the context of data analytics, a common pattern consists in analyzing the same data by means of different operators, which immediately leads to broadcasting communication in case the operators are deployed on different nodes of a distributed platform. As a concrete example, in the Apache Storm [109] framework for tuple processing, broadcasting is the only available model for data distribution.

Starting from this generic scenario, it can be observed that, by increasing the number of output processors, p will consume more and more computational resources to distribute the data to all the processors. Therefore, we propose *active communicators* as a general approach for alleviating the depicted phenomenon. As illustrated in Fig. 5.5, an active communicator is a GAM (sub-)net that behaves as a regular (passive) communicator, in that it routes pointers among the processors. Internally, an active communicator consists of multiple processors, linked by intermediate communicators according to some networking topology (e.g., tree), that actively collaborate to deliver each pointer.



(A) A simple net with a broadcasting communicator C.

(B) The net in (A) after replacing C with an active communicator, highlighted in gray.

FIGURE 5.5: Replacement of a broadcasting communicator with an active communicator shaped as a tree of depth 1.

We remark that, differently from the `svc` functions considered so far (e.g., cf. Sect. 5.3.1), the `svc` functions for the processors within active communicators are likely to not access the data carried by the incoming pointers. Indeed, the simplest form of such functions amounts to a single call to the `emit` function. Therefore, when based on dispatching policies that do not access the data as well (as in the common case), active communicators provide scalable, lightweight communication across processors in a GAM net.

Summary

In this chapter, we introduced GAM nets, a dataflow-like parallel programming model based on GAM. Moreover, we presented a template-based C++ API and implementation of GAM nets, for programming GAM nets in terms of parallel stateful processors exchanging smart GAM pointers. Finally, we showed how two well-known stream-parallel programming patterns, namely farm and pipeline, can be implemented as GAM nets.

Chapter 6

Higher-Level Programming Models on top of GAM

In this chapter, we discuss how GAM can be exploited to build RTSs for different high-level parallel programming models. Although this chapter does not represent a fully developed contribution, it outlines possible exploitations of the GAM model and implementation presented so far. Therefore, this chapter should be regarded as a detailed proposal for future work.

In particular, we focus on implementing RTSs for *data parallelism* and *task parallelism*. From an architectural perspective, this amounts to implementing a RTS in terms of *stream parallelism*, that is the primary type of parallelism provided by GAM. This approach is extensively used in the implementation of parallel RTSs. For instance, several implementations of the data-parallel OpenMP API (cf. Sect. 2.2.4) consist in a thread *farm* (cf. Sect. 5.3.2). Similarly, FastFlow uses stream skeletons (i.e., farm and pipeline) to implement data-parallel operators (cf. Sect. 2.4.2). As for the realm of distributed systems, the Flink [79] framework for Big Data analytics relies on a streaming runtime for implementing batch processing.[1]

On the same line as the mentioned frameworks, we envision to rely on passing pointers—rather than data—as the basic mechanism for limiting performance overhead.

This chapter proceeds as follows. In Sect. 6.1, we propose accelerated data structures for supporting data-parallel programming. In Sect. 6.2, we discuss the GAM implementation of a framework for task-parallel programming.

6.1 Accelerated Data Structures

We propose to target data parallelism in terms of *accelerated data structures*. This approach, which stems from programming specialized hardware accelerators such as GPUs, has been proven by Drocco *et al.* [74] to be effectively applicable in the context of distributed-memory platforms.

We proceed by illustrating the mentioned approach for exploiting clusters in a data-parallel manner (Sect. 6.1.1). Then we depict an API based accelerated data structures and transformations (Sect. 6.1.2), together with a GAM-based implementation (Sect. 6.1.3).

[1] Batch processing is a generic name to denote finite datasets processing, by means of a combination of data-parallel operations.

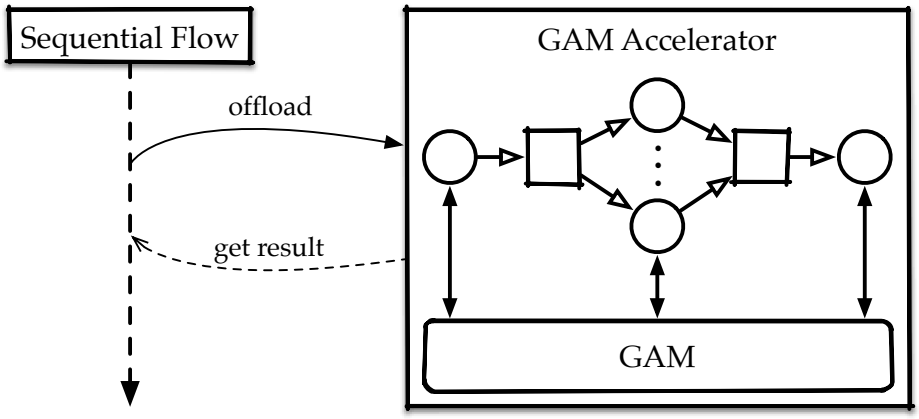


FIGURE 6.1: Cluster-as-accelerator paradigm with GAM.

6.1.1 Cluster-as-Accelerator Paradigm

The advent of specialized hardware accelerators imposed to find suitable programming models. A successful paradigm for programming accelerators is based on the concept of *offloading*. According to this approach, the user focuses on a *sequential execution flow*, defining only the sequence of operations to be applied to data. Each operation is offloaded to the external accelerator, where a specialized runtime takes care of executing the operation, possibly applying specific optimizations for the accelerator at hand.

From the programming model perspective, the depicted scenario leads to programs consisting in regular sequential code, enriched with two additional syntactic mechanisms:

- Functions for offloading computations to the accelerator;
- Accelerator-side computations, expressed in some specific language.

Among the most successful examples of accelerator programming, we consider the Nvidia CUDA framework. A CUDA program consists in plain sequential C++ code, endowed with invocation of kernels (i.e., accelerator-side computations). As for expressing kernels, CUDA provides a programming model in which the user specifies the computation to be performed by each *thread*, from within a multi-dimensional thread space. This model allows to easily express data-parallel computations (by mapping threads to atomic elements of a data structure) and is adopted by similar frameworks targeting accelerators, such as OpenCL.

Inspired by these frameworks, on the same line as Drocco *et al.* [74], we propose to target data-parallel programming by considering a whole cluster as hardware accelerator, to which data-parallel computations are offloaded from a sequential execution flow. We refer to this approach as the *cluster-as-accelerator* paradigm.

In the setting of GAM nets, offloading a computation to a cluster accelerator means triggering the execution of a GAM net, as depicted in Fig. 6.1.

6.1.2 C++ Library of Accelerated Containers

We propose to materialize the cluster-as-accelerator paradigm described in the previous section, into a C++ library, resembling the C++ STL, composed of *containers* and *transformations*. As for accelerator programming in the context of GPUs, an API based on a similar philosophy of STL-like containers and transformations is provided by the Nvidia Thrust [112] library.

We refer to the proposed containers as accelerated containers. For instance, the `transform` operation for the accelerated `vector` container resembles the recently introduced `transform` operation for C++ iterators. This operation is defined according to the following semantics, where a is the input container and f is an element-wise function:

$$\text{transform}\,(f, [a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)]$$

The `reduce` operation—that resembles the recently introduced `reduce` operation for C++ iterators—is defined according to the following semantics, where a is the input container and $\oplus$ is a commutative and associative pairwise function:

$$\text{reduce}\,(\oplus, [a_1, \ldots, a_n]) = a_1 \oplus \ldots \oplus a_n$$

From the programming model perspective, only f or $\oplus$ need to be specified to define, respectively, a `transform` or a `reduce` accelerator-side operation. The parameter functions f or $\oplus$, that we refer to as transformation *kernels*, are expressed in form of arbitrary C++ callable objects, provided that they do not rely on any local state (e.g., non-capturing lambda)

Listing 6.1 shows a minimal application that squares all the elements in an accelerated `vector`, where both accelerated containers and transformations are assumed to be enclosed in the `gal` namespace—which stands for GAM-accelerated library. Note that the proposed API is more similar to the Thrust API rather than the C++ STL, since the latter relies on *containers* (i.e., specific pointers) rather than whole collections. However, future C++ versions will introduce the *range* abstraction (i.e., logical views over sub-collections), thus introducing transformations with an API oriented to (sub-)collections.

Finally, although we did not consider the API for feeding data into (resp. extracting data from) an accelerated data collection from (resp. to) an external source (e.g., file), this can be easily imagined on the same line as the input-output functions provided by most mentioned frameworks for distributed processing, such as Spark. For instance, a vector could be filled by reading data from a file sitting in a distributed file system, accessible to both the sequential flow and the accelerator sides.

6.1.3 Implementation

From the viewpoint of the underlying GAM nets, an accelerated container consists in a set of smart global pointers, either public or private, possibly distributed among processors in the net. For instance, we consider as running example a simple run-time architecture, consisting in a master-workers pattern (cf. Sect. 7.1.4), in which the master maintains an *index* to associate a container to the pointers that compose it.

```
int main() {

    /* ... */

    gal::vector<int> a, b;

    /* add elements by calling push_back */

    /* ... */

    gal::transform(a, b,
        [] (int x) { return x * x; }
    );

    /* ... */
}
```

LISTING 6.1: Minimal application based on accelerated structures.

Since we rely on offloading from a sequential flow, the operations over accelerated containers are executed in a Bulk Synchronous Parallel (BSP) fashion: during the execution of an operation (i.e., the parallel phase in BSP), the involved pointers are spread among the processors, whereas at the beginning and at the end of the execution they are centralized under the master's control.

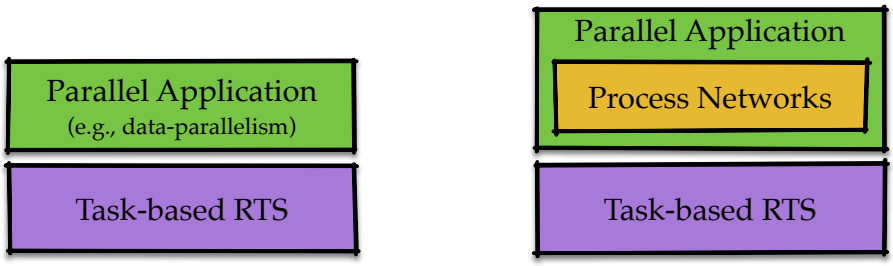
We remark that relying on pointers (rather than data) as the atomic components of accelerated containers leads to decoupling the logical ownership (i.e., the capability) over a data location from its physical location in memory. Although a pointer is free to flow among processors, the pointed memory is not moved unless the pointer is referenced by means of the `local` primitive (cf. Sect. 3.4.3). This aspect can be exploited, for instance, to design lightweight mechanisms for load balancing, in which only pointers are distributed (or redistributed) among processors.

Moreover, *locality-aware communicators* can be exploited to optimize the implementation according to the principle of *near-data processing*. At any time during the execution of an operation, the actual location of the memory pointed by a GAM pointer can be retrieved from the GAM runtime, which performs a local lookup to identify the author (cf. Sect. 3.4.2) for the slot. Therefore, considering the master-workers setting, it is possible to schedule each computation to a worker in such a way that memory transfers are minimized, for instance, because the worker is the author for the involved memory slot(s).

6.2 Task-based Parallel Programming

In this section we investigate different architectural layouts in the context of implementing a task-based parallel execution stack. Specifically, we discuss how GAM nets can be placed both above and below a task-parallel layer in the stack.

We proceed by discussing how tasks can be used to model any form of parallel processing (Sect. 6.2.1). Then we discuss different ways of implementing a task-based RTS on top of GAM nets (Sect. 6.2.2).



(A) General architecture of a task-based RTS.

(B) Task-based RTS with PN-based intermediation.

FIGURE 6.2: Tasks as universal parallel RTS.

6.2.1 Universal Model of Parallelism

Tasks are a pervasive concept in parallel computing. According to a common acceptation, tasks represent computations to be performed on some *input* data to produce some *output* data. This generic definition allows to to model any computation in terms of tasks, therefore, tasks can be regarded as a universal computation model. In particular, any parallel computation can be described in terms of the underlying *parallel activity graph*, that is, a graph in which nodes represent concurrent activities (i.e., tasks) and arcs represent dependencies between tasks. The parallelism arises implicitly since independent tasks (i.e., not linked by any dependency arc) can be performed in parallel.

Considering an hypothetical stack for executing parallel computations, based on the above principle, tasks sit below any form of parallelism. To complete the stack, a RTS must be provided, which actually executes parallel computations, represented in the form of tasks and their dependencies. In the following, we refer as a *task-based RTS* the combination of a task model and the associated RTS. Fig. 6.2 illustrates the task-based stack. In particular, Fig. 6.2B shows how tasks can be exploited to support the execution of Process Network (PN)-like parallel programs, including GAM nets.

As we discussed in Sect. 2.2.4, a number of frameworks have been proposed that couple a high-level programming model with task-based RTS. All those frameworks share the architecture depicted in Fig. 6.2A, whereas they vary in the parallel programming model they expose (i.e., the top level in Fig. 6.2A).

6.2.2 Implementing a Task-based RTS

Although, as discussed in the previous section, it is possible to represent any parallel computation in terms of task graphs, tasks cannot be placed at the bottom layer of the parallel execution stack, at least when considering conventional computing platforms. To be performed, a parallel computation eventually has to be matched with the existing parallel hardware, which in general consists of a finite set of inter-connected resources (e.g., the nodes of a cluster). Therefore, the complete picture for a task-based parallel execution stack, illustrated in Fig. 6.3, results from considering the

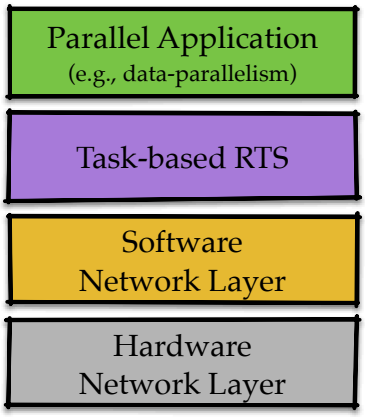


FIGURE 6.3: Complete task-based parallel RTS.

partial stack in Fig. 6.2 and adding an intermediate layer between the task-based RTS and the hardware layer, at the very bottom of the stack. Since the additional layer must be characterized by an internal structure that resembles the underlying hardware network (i.e., a finite set of inter-connected software nodes), we refer to this layer as Software Network Layer (SNL).

In the following, we present some common *organizations* for SNL networks and we discuss how they can be implemented as GAM programs (e.g., GAM nets).

Homogeneous SNL organization

The most straightforward network organization is the *complete graph*, in which each SNL node can communicate with any other node via a direct link. We refer to this organization, depicted in Fig. 6.4A, as *all-to-all*.

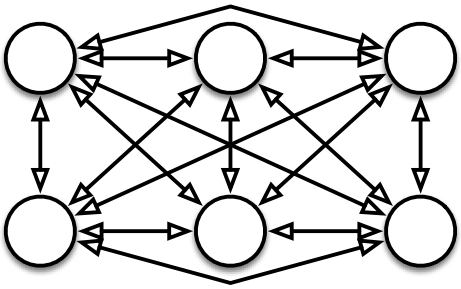
This simple organization is commonly exploited to design a *homogeneous* task-based RTS, in which all nodes are equivalent, in the sense that they play the same role. According to homogeneity, a task can be scheduled for execution on *any* node and the task graphs are cooperatively maintained by the nodes to determine which task(s) can be executed. To this aim, some distributed protocol is needed for coordinating the access to task graphs, since they are regarded as distributed data structures.

Usually, scheduling is coupled with *work stealing* [81], a mechanism to provide load balancing in case of dynamic workloads (i.e., fork/join execution model). In particular, two strategies for work stealing are commonly adopted, namely child stealing and parent stealing (aka. continuation stealing), depending on which branch of the fork is made available for stealing.

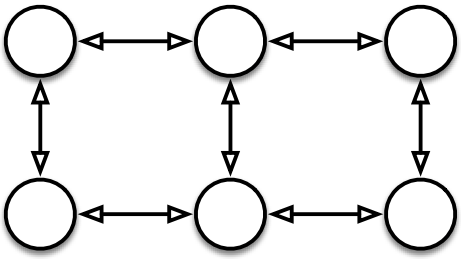
Moreover, the concept of *affinity* is exploited to ensure locality-aware scheduling. For instance, if a new task t' is scheduled by node n as side effect during the execution of another task t,[2] then t' should preferably be scheduled on n, at least in the case where t' accesses the same data as t.

By following the same approach as we discussed for realizing an all-to-all task-based SNL, a number of homogeneous organizations can be considered, in order to match the underlying hardware network. For instance, a *mesh* organization is shown in Fig. 6.4B.

[2]This scenario is also referred to as *dynamic* task graph.



(A) An all-to-all SNL organization with six nodes.



(B) A mesh SNL organization with six nodes.

FIGURE 6.4: Homogeneous SNL organizations.

As a concrete example of the depicted concept, we consider the implementation of *StarPU* [31], a task-based RTS supporting heterogeneous platforms, on top of which a number of applications have been implemented [34, 3]. To schedule tasks over a distributed platform, StarPU relies on MPI communications primitives [30]. In this case, the whole program is translated into an MPI program, which is conceptually analogous to a homogeneous SNL in which each node (i.e., a MPI rank) communicates with (a subset of) all other nodes.

Similarly, any PGAS framework (cf. 2.2.3) providing some task-based abstraction relies on the same mechanism. For instance, remote procedure calls (i.e., `async`) in UPC++ [131] and HPX [90] are implemented by means of an underlying communication subsystem (i.e., a SNL), namely GASnet and parcel, respectively.

GAM Implementation Let us consider the problem of implementing a homogeneous SNL in terms of GAM programs. First, we let tasks be represented by smart global pointers (cf. Ch. 4), as input and output data associated to the task. Scheduling a task means passing all the capabilities associated with the task—thus including the capabilities over the input and output data associated to the task—to the node on which the task is being scheduled. Note that, since each input value v is mapped to a GAM slot, which is in turn mapped to either a public or a private pointer, v is *always* accessed in either a read-only or a read-write manner, respectively.

Therefore, the access model for each input value can be regarded as an intrinsic attribute of the memory location.[3] Considering GAM nets, an extension is needed to the model discussed in Sect. 5.1.2, to support the scenario with multiple communicators attached to the same processor. With such an extension, each processor can be attached to a one-to-many communicator, with a simple dispatching policy that allows to deliver each task to the processor selected by the scheduling algorithm. Finally, the mentioned distributed protocols (e.g., for managing the distributed task graph) can be implemented by means of non-mappable GAM values, whereas affinity can be realized by locality-aware communicators, as discussed in Sect. 6.1.3.

Non-Homogeneous SNL organizations

Another common organization for task-based RTS is the *master-workers*. In contrast with homogeneous organizations, master-workers organization stem from considering nodes that play different roles with respect to the execution of task graphs.

In particular, the master holds exclusive knowledge and responsibility over the task graph to be executed. Moreover, it is the master's role to keep track of each task's state (e.g., not scheduled yet, running, completed, etc.), to determine which task(s) can be executed, according to the dependencies indicated by the graph. Conversely, workers execute tasks in a "blind" manner, without any awareness of the global state of the computation. Each worker simply receives one task from the master and, after having executed that task, notifies the master of its completion.

Relying on a master-workers organization has the attractive property, from the programmability perspective, of centralizing the scheduling logic in a single code entity (i.e., the master), thus simplifying the implementation of custom scheduling policies, based on the specific application at hand. For instance, widespread adoption of this organization can be observed in the context of RTS for data analytics frameworks, as we show in Sect. 7.1.4.

We regard master-workers organizations as a simple form of non-homogeneous organizations. Orthogonally to homogeneous organizations, in which all functionalities are distributed among all the nodes, non-homogeneous organizations promote code compartmentalization, by localizing some specific functionality (e.g., scheduling) at specific nodes (e.g., the master).

GAM Implementation In terms of GAM nets, a master-workers organization is a farm pattern with an additional feedback channel through which workers send notifications of task completions back to the master. We discuss this approach at the end of Sect. 7.1.4 in the context of RTS for data analytics, where a task graph is generated from a DAG of operations over an input data collection. We remark that active communicators (cf. Sect. 5.3.3) can be exploited to alleviate the drawback on performance caused by centralization, drawback intrinsic to non-homogeneous organizations.

[3] This desirable property, commonly referred to as *no-aliasing*, is widely exploited by compilers since it allows a number of optimizations. However, in that scenario, no assumptions can be made a priori on the access model for a given memory location, hence no-aliasing has to be explicitly guaranteed by the programmer.

Summary

In this chapter, we explored the exploitation of GAM programs—especially GAM nets—in the context of implementing RTSs for higher-level parallel programming models. Specifically, we discussed the implementation of data parallelism, by means of the cluster-as-accelerator paradigm, and the implementation of task parallelism, by exploring different organizations for GAM programs at the basis of task-based RTSs.

Chapter 7

Experimental Evaluation

In this chapter, we show the effectiveness of the approach discussed in this thesis, with respect to two fundamental dimensions: the *expressiveness* carried by the proposed API, along with the associated parallel programming models, and the *performance* sustained by the stacked implementation.

To the first aim (Sect. 7.1), we focus on porting existing shared-memory applications to GAM nets. In particular, we highlight the effectiveness of relying on *pointers* as a useful programming abstraction for reducing the gap between the shared-memory and the distributed-memory programming models.

To the second aim (Sect. 7.2), we evaluate the performance exhibited by two *stream processing* applications, since they represent the most natural target for GAM nets. Although we discussed how we expect smart GAM pointers to impact scalability (Sects. 4.1.1 and 4.2.1 for public and private pointers, respectively), we are not able to confirm experimentally the speculated behaviors. Therefore, we regard this task as the most urgent future work.

7.1 Expressiveness

In this section, we demonstrate the expressiveness of GAM nets by showing the implementation of some use cases, from the following application domains: video processing (Sect. 7.1.1), financial data processing (Sect. 7.1.2), systems biology simulation (Sect. 7.1.3), and data analytics (Sect. 7.1.4).

7.1.1 Two-Phase Video Restoration

In the field of signal processing, in particular image and video processing, computations are commonly represented as a chain of successive operators (or phases), that progressively transform input signals into filtered signals.

For instance, Aldinucci *et al.* [23] proposed a two-phase filter for removing "salt and pepper" noise, together with a shared-memory implementation exploiting both multi-core and GPU parallelism for restoring a single image. The filter was extended by Aldinucci *et al.* [21] to target video signals (i.e., ordered streams of images), possibly exploiting different GPUs for different frames.

In the same vein, we consider the platform generally known as cluster of GPUs, composed of a network of identical hosts, each attached to a GPU.[1] We can exploit the depicted platform, in terms of GAM nets, by means of

[1] For the sake of simplicity, we do not treat the case of multiple GPUs attached to a single host, but such a generalization would be straightforward.

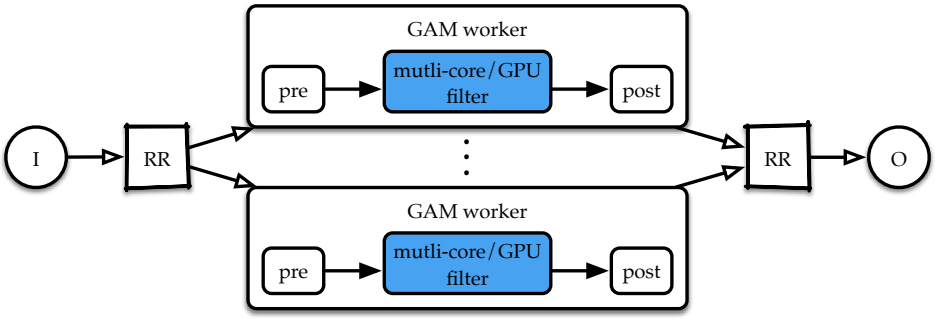


FIGURE 7.1: GAM net for video restoration on clusters of GPUs.

an *ordering farm*, in which each frame of the input video is processed by one of the GPU-accelerated hosts. The ordering semantics is embedded into the farm communicators, by letting both the distributing and the gathering communicators (respectively, D and G in Fig. 5.4) agree on the sequence according to which workers are selected for, respectively, pushing and pulling frames.

The depicted GAM net is illustrated in Fig. 7.1. A source processor is deputed to provide the input video by reading frames from a source, such as a camera or a file. Each frame is assigned to a worker processor, according to the round-robin distribution provided by the distributing communicator. When a frame pointer is delivered to a worker, the pointer is accessed and its content is pre-processed (the "pre" box) in order to make it compatible with the existing shared-memory filter (highlighted in blue). For instance, if each frame is represented and exchanged at GAM level as a contiguous memory chunk, it would be necessary to wrap each chunk into a data structure of the type requested for interfacing with the shared-memory filter. Dually, after being filtered, each frame is post-processed (the "post" box), in order to extract a GAM-friendly representation, and sent downstream via a gathering communicator, that maintains the original order as the input stream, as discussed above. Finally, a sink processor is deputed to manage the output stream, for instance by storing it to a file.

7.1.2 High-Frequency Stock Option Pricing

The analysis of financial data by means of stream processing is a long standing approach. In particular, we consider the scenario in which the input data enter the processing system at a high rate, which results in processing *high-frequency* financial data. For example, this problem has been formulated in terms of the farm pattern (cf. Sect. 5.3.2) by De Matteis *et al.* [66]. Similarly, Misale [105] implemented a filter for financial streams on top of the PiCo framework (cf. 7.1.4), using mainly `map` and `reduce` operators.

Along the same line as the mentioned proposals, we implemented a *n*-worker GAM farm that, given an input stream of stock options represented as tuples that include the stock name, computes the price of each option by

calculating the well-known Black–Scholes formula.[2] We refer to this benchmark as Stock Option Pricing (SOP). We implemented the farm in such a way that the options for a given stock (identified by the stock name) are all processed by the same worker. Although not strictly required—unless some ordering constraint is imposed—this partitioned processing allows to generalize the workers from functions (thus stateless) to arbitrary, possibly stateful operators. To this aim, we exploited the key-partitioning communicator (BKT in Fig. 7.4a) as scheduler-side communicator, by passing the stock name upon calling the `emit` function (cf. Sect. 5.2.2) from the scheduler.

7.1.3 CWC Systems Biology Simulator

In recent years, a growing number of systems biology problems have been tackled by means of *simulation*, that provides accurate results but at the expense of considerable computational costs. Moreover, costs increase even further when the complete simulation-analysis workflow is considered.

In this context, Aldinucci *et al.* [7] proposed a FastFlow implementation, targeting multi-core platforms, of a simulation-analysis workflow for systems biology, based on the CWC formalism by Coppo *et al.* [58].

We propose an implementation of the simulation-analysis workflow in terms of GAM net, as illustrated in Fig. 7.2. A source processor (S) generates and distributes n random seeds, where n is the number of simulation instances to be performed. Each simulation worker executes a simulation engine, that outputs a stream of simulation outcomes, sampled at a fixed rate τ.

The aligner processor gathers the samples according to a `FromAny` gatherer (FA) and produces as output a stream of *snapshots*. The k-th snapshot, with $k > 0$, that we denote as $s(k)$, is the set of all the simulation outcomes sampled at $k\tau$. Therefore, it corresponds logically to the following tuple, where $s_i(k)$ is the k-th outcome of the i-th simulation instance (i.e., the outcome sampled at time $k\tau$):

$$s(k) = \{s_1(k), \ldots, s_n(k)\}$$

The analysis of simulation samples is performed by statistical workers in a *sliding window* manner. For each sampling time $k\tau$, a number of statistical filters is computed over a time-neighborhood of $s(k)$, that we denote as $\sigma(k)$, defined as follows, where w is the width[3] of the window:

$$\sigma(k) = \left[s\left(k - \frac{w}{2}\right), \ldots, s(k), \ldots, s\left(k + \frac{w}{2}\right)\right]$$

The sliding behavior arises since, given two consecutive sampling times $k_1\tau$ and $k_2\tau$, the two corresponding windows overlap by sharing all the elements but the extreme ones:

$$|\sigma(k_1) \cap \sigma(k_2)| = w - 1$$

[2] This benchmark is the GAM implementation of the *blackscholes* benchmark from the PARSEC suite [35].

[3] The width of a window is an even number that represents the number of items that falls in the window, in addition to the item at the center of the window, thus $w = |\sigma(k)| - 1$.

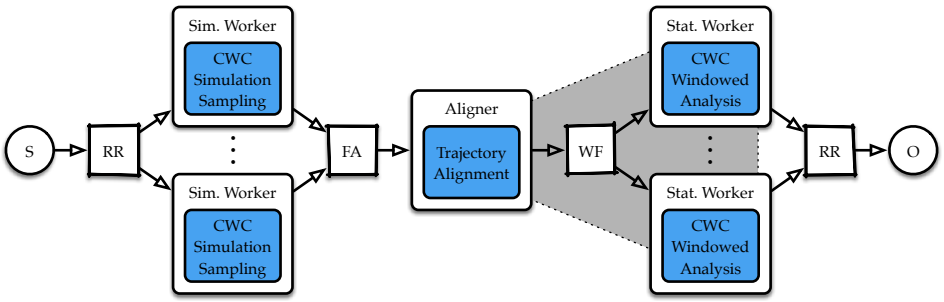


FIGURE 7.2: GAM net for the CWC simulation-analysis workflow.

This form of windowed processing has been implemented in a farm-based setting by De Matteis *et al.* [67]. In the proposed implementation, denoted as *Window-Farming* (WF), each worker maintains a windowed view over the stream generated by the aligner, where windows across different workers may overlap. Therefore, each snapshot $s(k)$ is shared by a set of workers, namely, those building at least a window that includes $s(k)$.

The implementation of WF in terms of the farm pattern, from the perspective of effective memory management, has been studied by Torquati *et al.* [126], where C++ shared pointers are exploited to couple the depicted sharing mechanism with automatic memory management. In the same vein, we implement WF in terms of a GAM farm pattern (the dotted gray triangle in Fig. 7.2), relying on public pointers (cf. Sect. 4.1) to represent snapshots, since they are shared by different workers. Moreover, we encode the WF distribution logic in the *multicast* communicator linking the aligner to statistical workers (WF). On the same line as the implementation proposed by Torquati *et al.* [126], each snapshot is delivered by WF to all the statistical workers that build at least a window that includes $s(k)$.

Finally, since WF realizes a round-robin distribution of windows among statistical workers, the same mechanism discussed in Sect. 7.1.1, based on gathering according to a round-robin communicator (RR), can be exploited to preserve the order among samples.

We remark that the proposed approach can be generalized to target any similar simulation-analysis workflow, since it treats all the CWC-specific logic as black boxes (highlighted in blue in Fig. 7.2), for both the simulation and the analysis components.

7.1.4 PiCo Data Analytics Framework

In the context of applications for data analytics, with particular focus on the realm of so-called Big Data processing, a number of frameworks have recently been proposed (e.g., Google MapReduce [68], Apache Spark [128, 129], Apache Storm [109], Apache Beam [4, 33]), in which processing is expressed in terms of APIs with different flavors.

Misale *et al.* [107] proposed a unifying model, based on the dataflow model, according to which any application expressed in one of the mentioned frameworks can be formulated, at abstract level, in terms of a graph of functional-style operators. On top of the unifying model, Misale [105] proposed PiCo, a DSL, whose semantics has been formalized by Drocco *et*

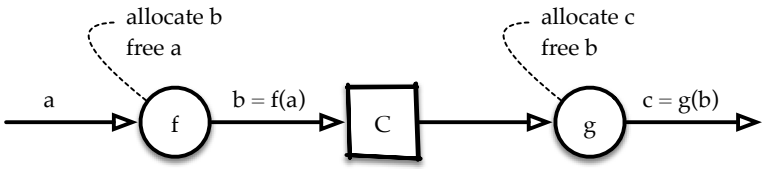


FIGURE 7.3: Decoupled allocation and freeing of PiCo micro-batches.

al. [76], endowed with a C++ implementation. In the proposed implementation, shared-memory parallelism is exploited by mapping an application to a composition of FastFlow farms and pipelines. In particular, the farm pattern is exploited to express the parallelism both *between* operators (e.g., computing multiple statistic measures on the same data) and *within* an operator (e.g., parallel implementation of the `map` function). Data collections to be processed, either (bounded) data sets or (unbounded) streams, are split into *micro-batches*, that are streamed to the application as indivisible items.

Replicating the depicted approach in terms of GAM nets is straightforward, since farms and pipelines can be expressed (cf. Sects. 5.3.2 and 5.3.1, respectively). In particular, the automatic memory management provided by smart global pointers (cf. 4.3) reduces the complexity arising from decoupling allocation and freeing of micro-batches, as depicted in Fig. 7.3. In general, a micro-batch (e.g., *b* in the figure) is allocated by the first operator that outputs it (*f* in the figure), whereas it gets freed by the downstream operator that processes it (*g* in the figure) to produce another micro-batch.

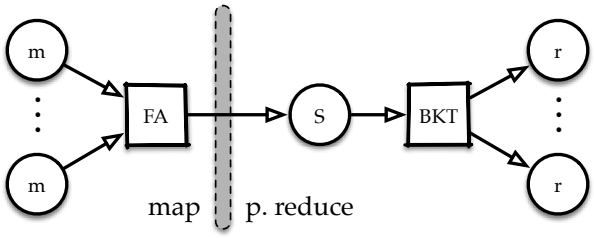
Partitioning

A common scenario for data analytics applications consists in dealing with data collections representing multiple sub-collections, in the form of partitioned collections. For instance, considering tuple processing, such a partitioning is commonly defined by grouping tuples carrying the same value at some field of the tuple, generally referred to as the *key*.

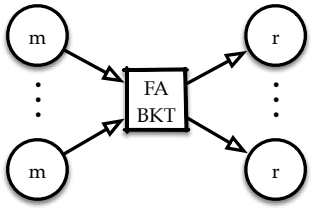
In the context of GAM nets for data analytics, a partitioning-aware operator can be implemented as a farm, in which the distribution communicator is based on the `ByKeyTo` dispatcher (cf. Tab. 5.1). In this way, all the tuples belonging to the same partition are delivered to the same processor, providing two desirable effects. First, the depicted schema simplifies supporting a *partitioned state* (i.e., a load/store memory slot for each group), since precluding tuples from the same group to be processed by different processors guarantees that no concurrency occurs over the state between different processors. Second, this schema preserves the order among items of the same group, thus eliminating the need for enforcing it, for instance by an ordering farm.

Shuffling

In addition to the discussed implementation, based on farms and pipelines, Misale [105] proposed graph refactoring as a general technique for optimizing execution graphs, e.g., by removing points of centralization.



(A) Naive GAM net for `map` followed by partitioning-aware `reduce`.



(B) The net from (A), in compact form.

FIGURE 7.4: Compacting a GAM net by introducing a shuffling communicator.

For instance, the graph for a `map` operator followed by a partitioning-aware `reduce` operator can be compacted, as shown in Fig. 7.4, in terms of GAM nets. In the example, the sequence composed by the gathering communicator FA (based on a `FromAny` pulling dispatcher), the scheduler processor S, and the distributing communicator BKT (based on a `ByKeyTo` pushing dispatcher) is collapsed into a single communicator (based on a pulling `FromAny` and a pushing `ByKeyTo` dispatchers, respectively).

In the resulting data distribution schema, each data item coming from one of the "left-hand side" processors is delivered to one of the "right-hand side" processors, according to the partitioning logic. This schema, commonly referred to as *shuffling*, is a common pattern in the context of data analytics. For instance, a MapReduce application [68] consists of two successive steps, referred to as Map and Reduce, respectively. However, between these two steps, an additional Shuffle step is performed, where worker nodes redistribute data produced by the Map step, based on the output keys, such that all data with the same key are processed by the same Reduce worker node.

Master-Workers

The design principle underlying PiCo implementation consists in mapping the components of data analytics application to a low-level network, whose nodes are mapped in turn to computational resources. An orthogonal approach, adopted by a number of existing frameworks, consists in representing a data analytics application as DAGs of operations and treat the DAG as a *task graph*, which is a set of computations connected by data dependencies. The task graph is then executed by a master-workers RTS, that is, a low-level network that acts as an interpreter.

The master-workers approach for data analytics can be implemented in terms of a GAM farm, as depicted in Fig. 7.5. With respect to the generic

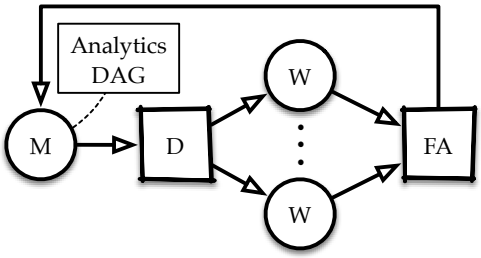


FIGURE 7.5: GAM master-workers RTS for data analytics.

farm pattern discussed in Sect. 5.3.2, in a master-workers DAG interpreter, the farm scheduler and collector (i.e., the master components) collapse into the same processor, that we refer to as the master. The master processor (M) holds the exclusive access and responsibility over the data analytics DAG to be executed, whereas workers (W) execute tasks as instructed by the master. The task scheduling policy, that should be designed according to the underlying networking platform, is encoded in the dispatching communicator (D), whereas task gathering is likely to be implemented as a simple `FromAny` communicator (FA).

In Sect. 6.2, we provide some further details about how task-based RTSs can be implemented in terms of GAM nets.

7.2 Performance

In this section, we provide a preliminary evaluation of the performance sustained by our model and its C++ implementation. To this aim, we measured the performance of two applications (the Two-Phase Video Restoration and the High-Frequency Stock Option Pricing in Sect. 7.1), when executed on three small clusters and we characterize them in in terms of *relative speedup* as well as *absolute speedup* (with respect to baseline implementations).

This section proceeds as follows. In Sect. 7.2.1, we detail the experimental setting. In Sect. 7.2.2, we report the measured performances. Finally, in Sect. 7.2.3, we discuss the observed results.

7.2.1 Setting

The following platforms have been used for our experimental evaluation:

- The Open Computing Cluster for Advanced data Manipulation supercomputer (OCCAM) [127, 6] was designed and is managed by the University of Torino and the National Institute for Nuclear Physics. For our experiments, we used 4 GPU nodes from OCCAM, equipped with 2x sockets of 12-core Intel Xeon Processor E5-2680 v3 @2.1Ghz, 128GB (8x16GB) DDR4 RAM, and 2x Nvidia K40 on PCI-E Gen3 x16.

- The *Research92* cluster (r92) was designed and is managed by the IBM T.J. Watson Research Center (NY). For our experiments, we used 8 nodes from r92, equipped with 2x sockets of 10-core IBM POWER8 Processor @3.5Ghz, 256GB 8x Memory Riser (4x8GB) DIMMs, and 1x Nvidia GP100 SXM2 on PCI-E Gen3 x16.

- The *Paradigm* cluster at the Computer Science department of Torino. It is composed of 4 low-cost low-energy nodes, equipped with an Intel Atom Processor C-2750 @2.40 GHz and 16GB RAM.

All the considered platforms are small-scale representations of common clusters from the HPC world. In particular, OCCAM and r92 represent the architecture referred as "clusters of GPUs", in which multiple nodes, each attached to one or more GPU accelerators, are interconnected by means of some networking hardware. The paradigm cluster represents a common architecture among clusters for data processing, characterized by a large number of commodity workstations.

To demonstrate the flexibility with respect to the networking hardware, enabled by implementing the GAM stack on top of libfabric, we select a different network fabric among those available on each considered platforms. Specifically, we selected InfiniBand (i.e., the *verbs* libfabric provider) on OCCAM, Ethernet (i.e., the *sockets* libfabric provider) on r92, and A3Cube RONNIEE (i.e., the experimental *dpa* libfabric provider, developed by Inaudi [86]) on Paradigm.

For all the measurements in the following, we report the median value of 100 observations. For simplicity, we omit any additional statistical information, since all the measurements exhibited negligible variance.

7.2.2 Results

Two-Phase Video Restoration

In Sect. 7.1.1, we discussed the implementation, in terms of GAM nets, of the two-phase video restoration filter proposed by Aldinucci *et al.* [21]. In particular, from the existing shared-memory implementation, supporting both multi-core and GPU acceleration, we derived a GAM net implementation that can be deployed on a cluster of multi-core nodes, each attached to a number of GPUs.

The proposed implementation, illustrated in Fig. 7.1 (p. 100), is based on a GAM ordering farm, whose n worker processors embed the shared-memory implementation, that in turn exploits m GPUs for each worker. Each processor in the farm (i.e., a scheduler, n workers, and a collector) is deployed on one of the cluster's node, thus using $n + 2$ nodes. When the number of processors exceeds the number of available nodes, multiple processors gets mapped to the same node. In the following, we refer to n as the *parallelism degree*.

We executed the filter over short video streams of 2048 frames at two *resolutions*, namely VGA (i.e., 640×480) and 720p (i.e., 1280×720). For each resolution, we considered two levels of *noise* corrupting the input stream, namely low (30%) and high (70%), representing the relative amount of corrupted pixels.

We consider the *throughput*, measured in filtered frames per second, as the reference performance metric. Denoting by t_n the throughput supported by the configuration with n workers, we define the *relative speedup* as follows:

$$\sigma(n) = \frac{t_n}{t_1} \tag{7.1}$$

Figs. 7.6 and 7.7 show the relative speedup exhibited by the GAM filter, with respect to the number of worker processors, on the OCCAM and r92 clusters, respectively. In both cases, the filter has been configured to exploit one GPU for each worker (i.e., $m = 1$).

To evaluate the true performance gain, thus taking into account the overhead imposed by the GAM runtime, we also consider the *(absolute) speedup*. Given a reference implementation, that we refer to as the *baseline*, we define the absolute speedup over the baseline as follows, where t_b is the throughput supported by the baseline:

$$s(n) = \frac{t_n}{t_b} \tag{7.2}$$

For the video restoration filter, we used the existing (parallel) shared-memory application by Aldinucci *et al.* [21] as the baseline implementation, written in FastFlow and also exploiting GPU acceleration. Figs. 7.8 and 7.9 show the absolute speedup exhibited by the GAM filter, with respect to the number of worker processors, on the OCCAM and r92 cluster, respectively. In both cases, the filter has been configured to exploit one GPU for each worker (i.e., $m = 1$).

Finally, Fig. 7.10 shows the maximum throughput observed for the filter, exploiting all the available parallelism, including multiple GPUs for each worker (i.e., $m = 2$) on the OCCAM cluster.

High-Frequency Stock Option Pricing

We executed SOP over the *native* input dataset from PARSEC, to which we added a random stock name to each option. As in the previous benchmark, we consider the supported throughput, measured in processed tuples per second, as the reference performance metric.

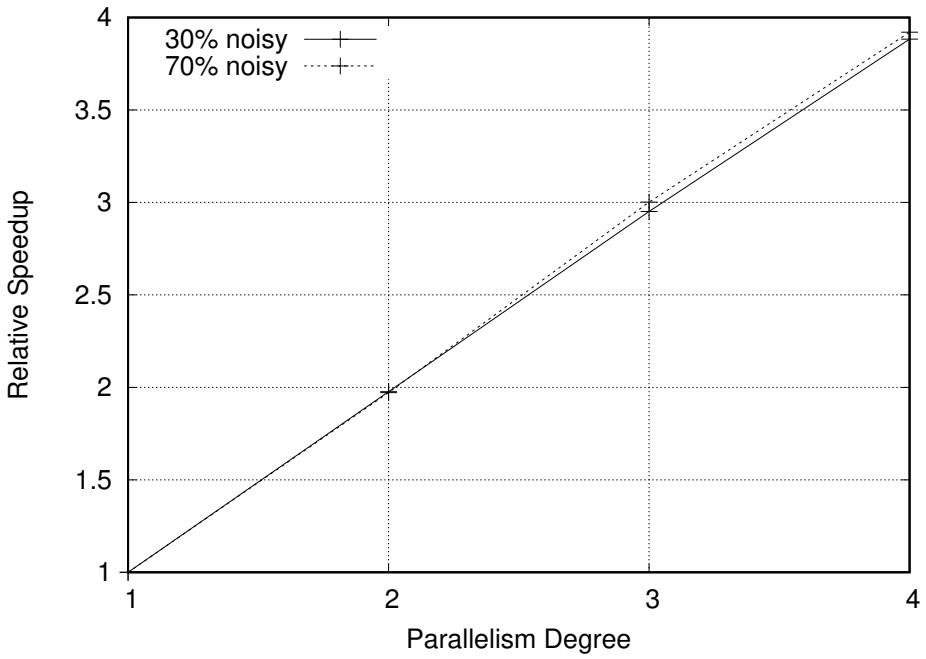
Figs. 7.11 and 7.12 show the relative speedup exhibited by SOP, with respect to the number of worker processors, on the OCCAM and Paradigm clusters, respectively.

To evaluate the performance in terms of absolute speedup (Eq. 7.2), we consider as baseline a shared-memory FastFlow implementation that exploits pipelining to hide the latencies induces by reading and writing the stock options stream. Figs. 7.13 and 7.14 show the speedup exhibited by SOP, with respect to the number of worker processors, on the OCCAM and Paradigm clusters, respectively.

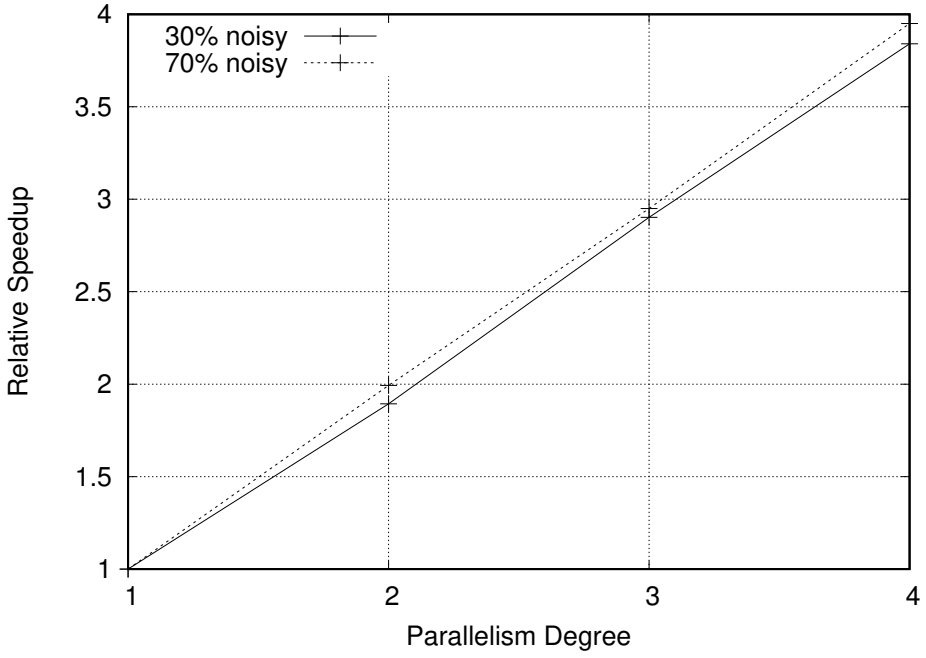
7.2.3 Discussion

Two-Phase Video Restoration

The GAM filter exhibits almost linear relative speedup, for both VGA and 720p resolutions, on the OCCAM cluster (Fig. 7.6). Relative speedup is sub-linear on the r92 cluster (Fig. 7.7), arguably due to the known limitations in the Ethernet providers for libfabric. As for absolute speedup (Figs. 7.8 and 7.9), on both platforms it is almost identical to the relative speedup, thus demonstrating that the overhead introduced by GAM nets for this benchmark is negligible.

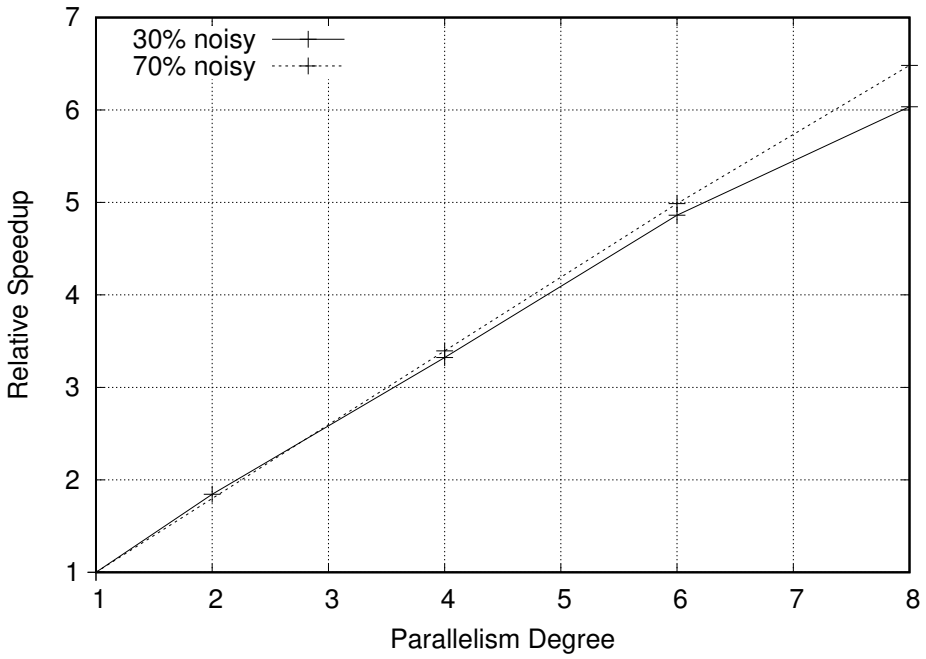


(A) VGA resolution.

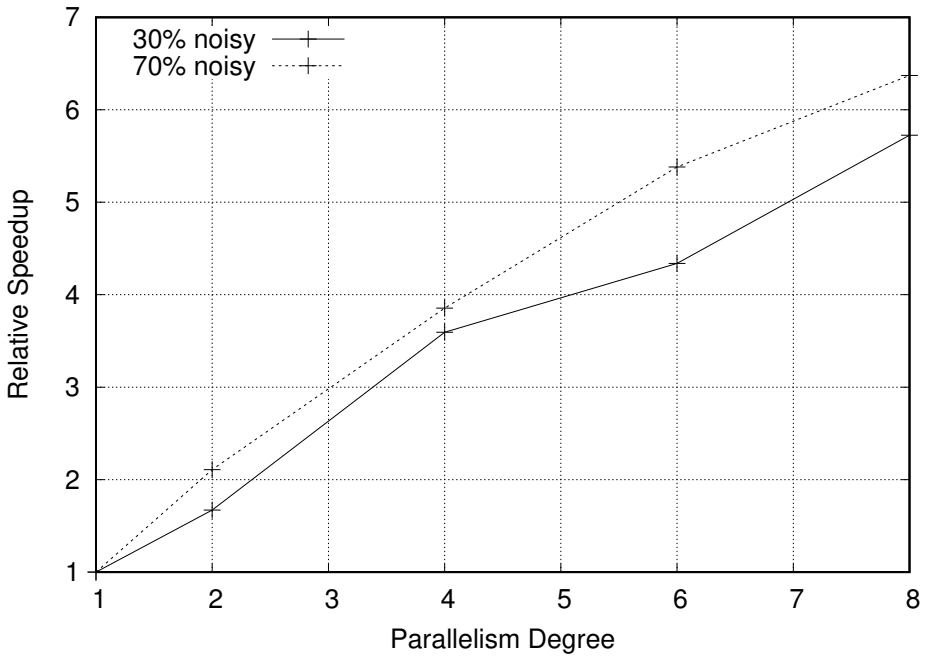


(B) 720p resolution.

FIGURE 7.6: Relative speedup of GAM video restoration filter on four GPU nodes from the OCCAM cluster, with a Nvidia K40 GPU per node.

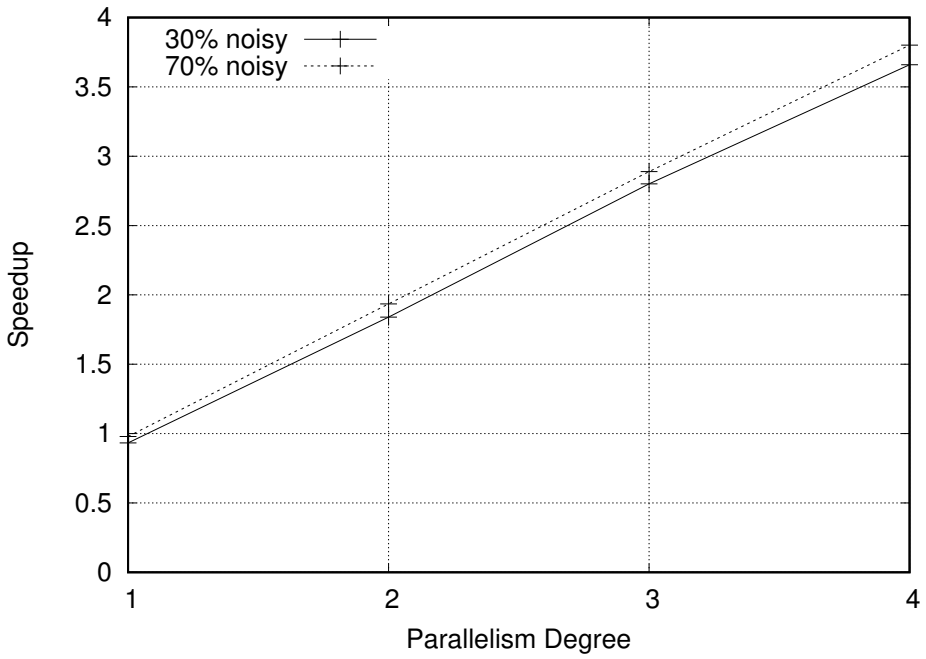


(A) VGA resolution.

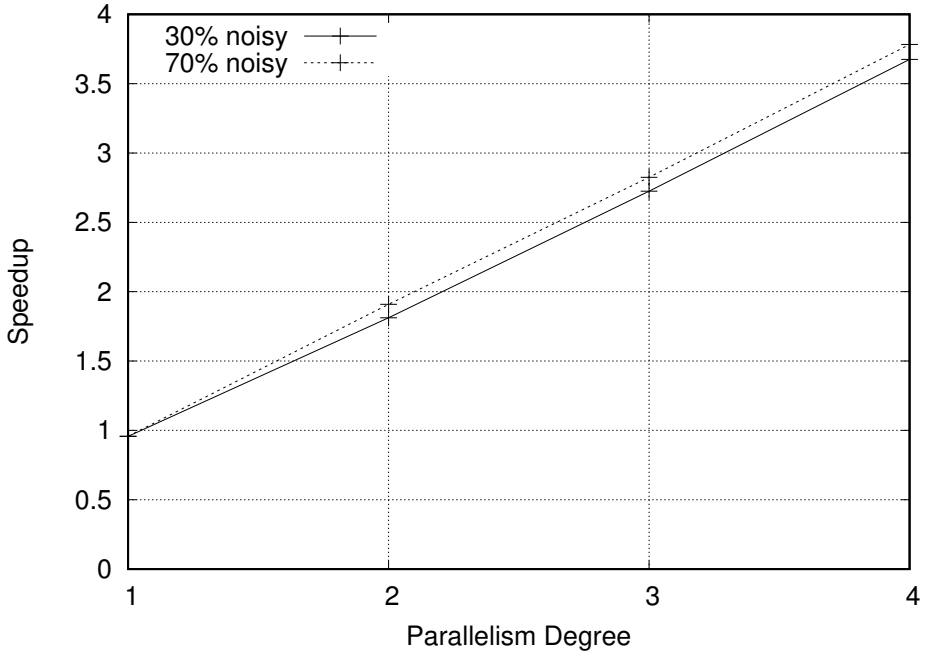


(B) 720p resolution.

FIGURE 7.7: Relative speedup of GAM video restoration filter on eight nodes from the r92 cluster, with a Nvidia GP100 GPU per node.

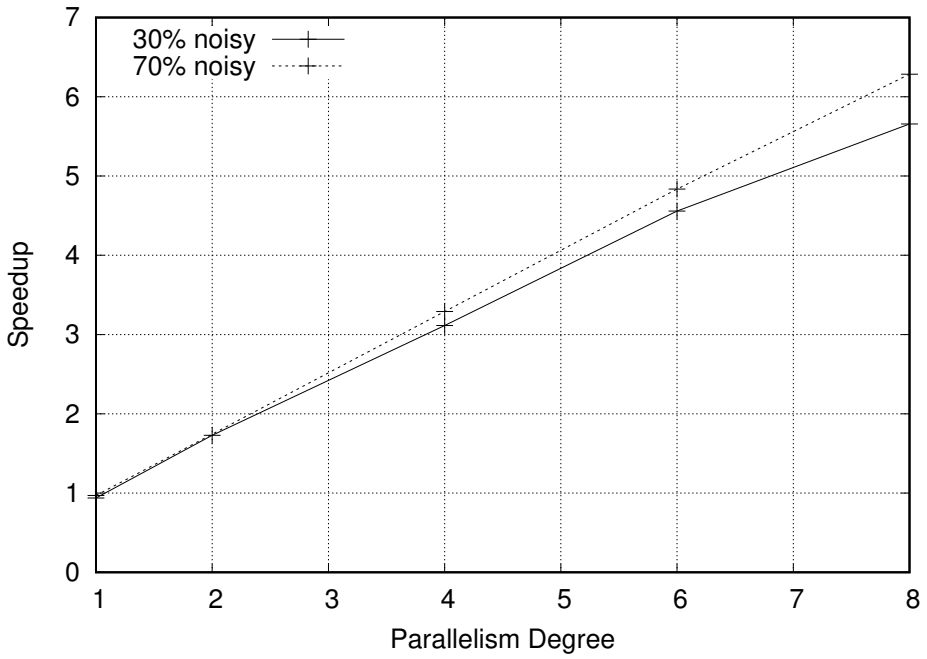


(A) VGA resolution.

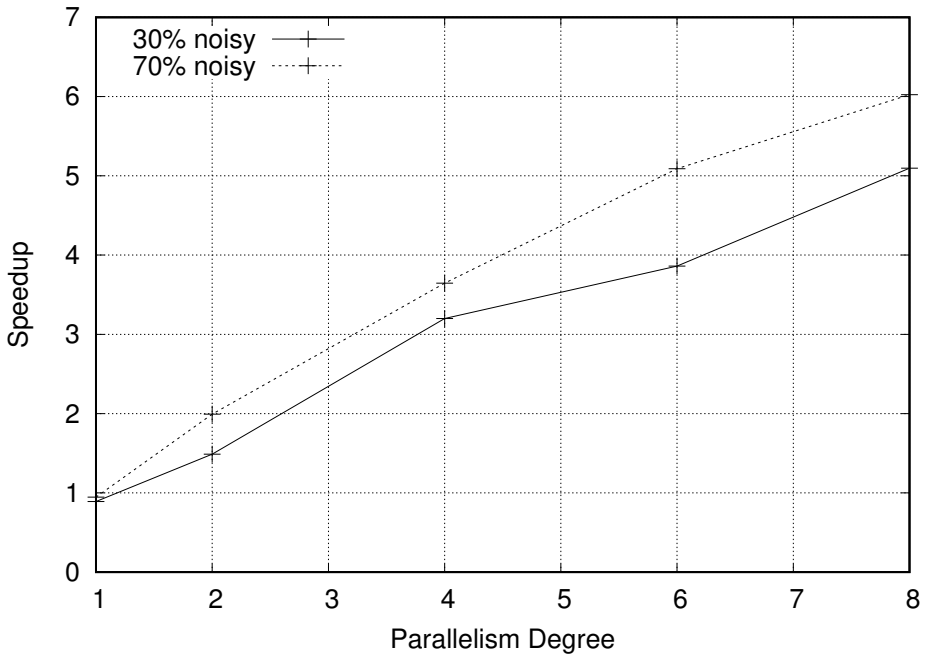


(B) 720p resolution.

FIGURE 7.8: Absolute speedup of GAM video restoration filter on four GPU nodes from the OCCAM cluster, with a Nvidia K40 GPU per node.

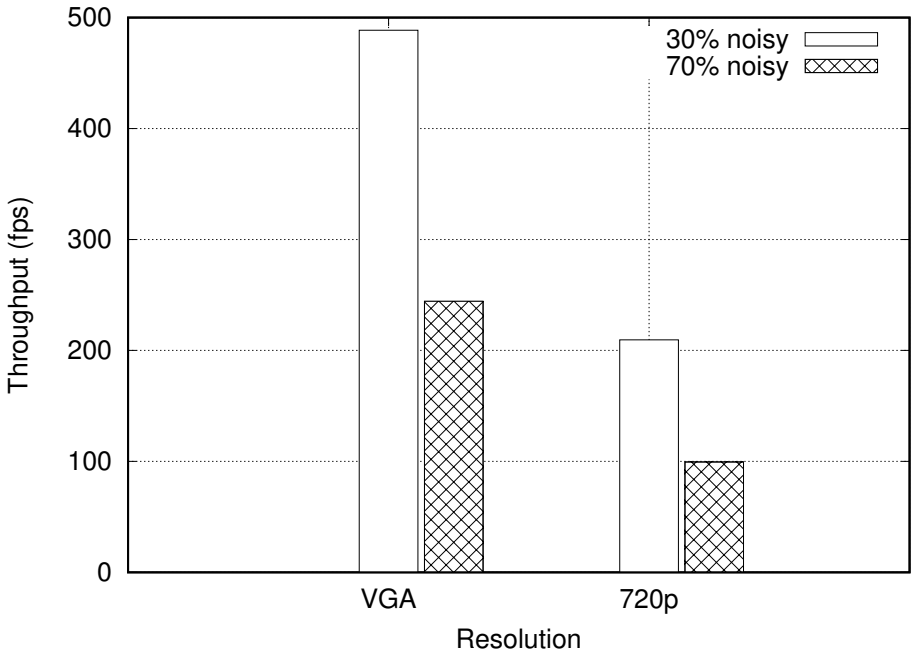


(A) VGA resolution.

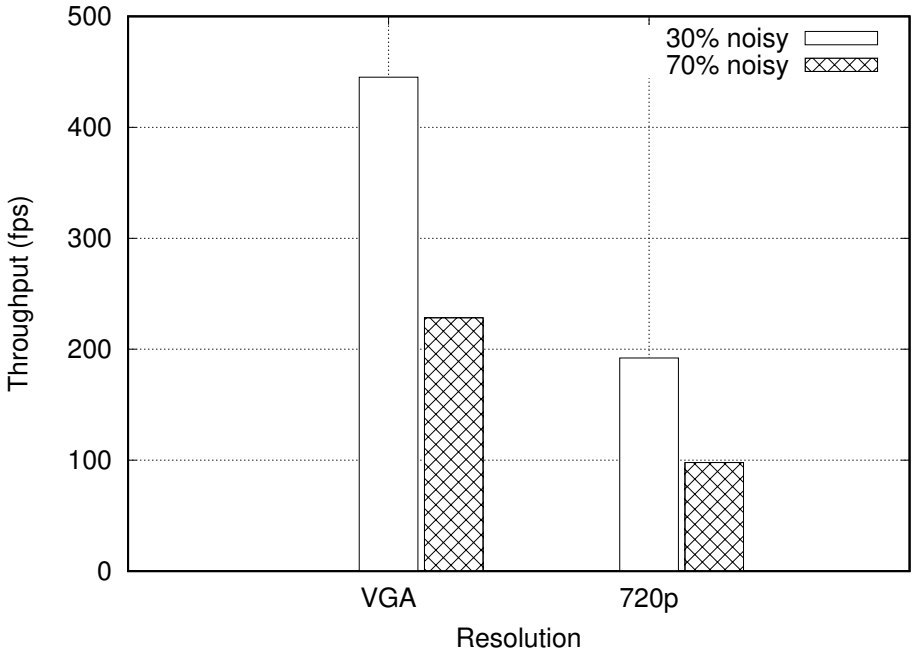


(B) 720p resolution.

FIGURE 7.9: Absolute speedup of GAM video restoration filter on eight nodes from the r92 cluster, with a Nvidia GP100 GPU per node.



(A) Four GPU nodes from the OCCAM cluster, with 2x Nvidia K40 GPUs per node.



(B) Eight nodes from the r92 cluster, with a Nvidia GP100 GPU per node.

FIGURE 7.10: Maximum performance of GAM video restoration filter, with the maximum number of nodes for each cluster, and the maximum number of GPUs for each node.

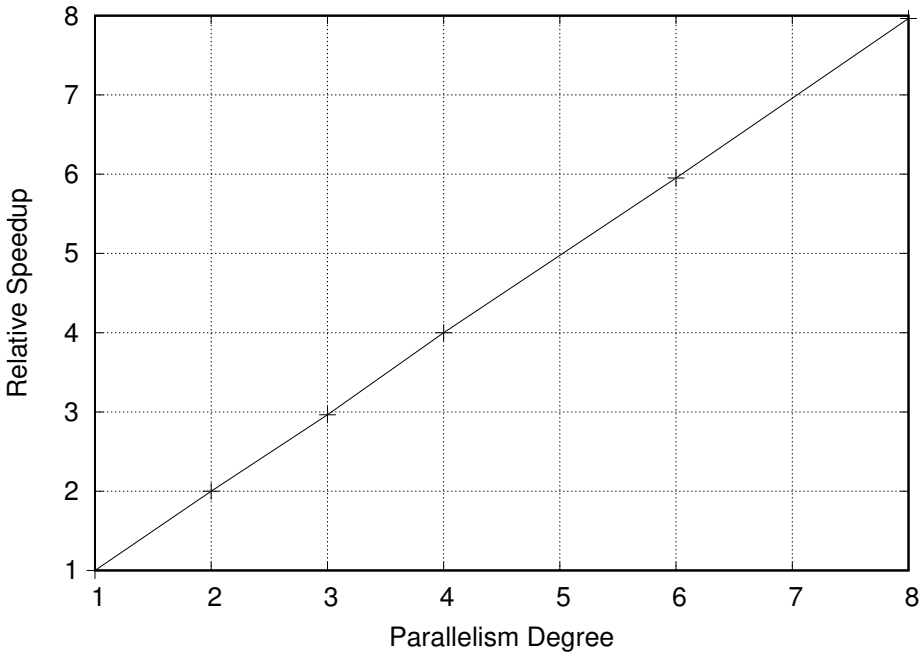


FIGURE 7.11: Relative speedup of SOP on four GPU nodes from the OCCAM cluster.

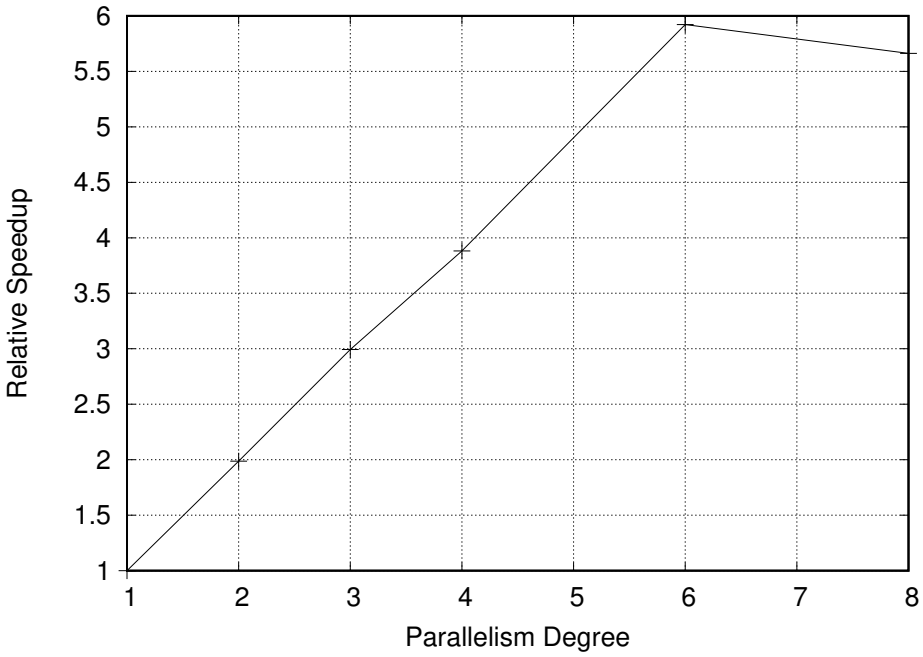


FIGURE 7.12: Relative speedup of SOP on the four-node Paradigm cluster.

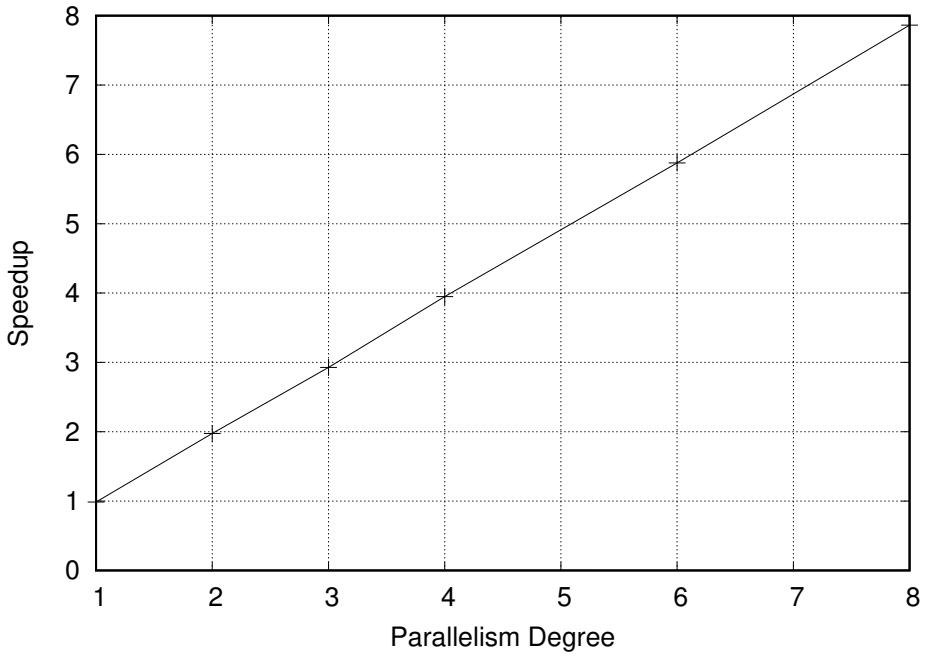


FIGURE 7.13: Absolute speedup of SOP on four GPU nodes from the OCCAM cluster.

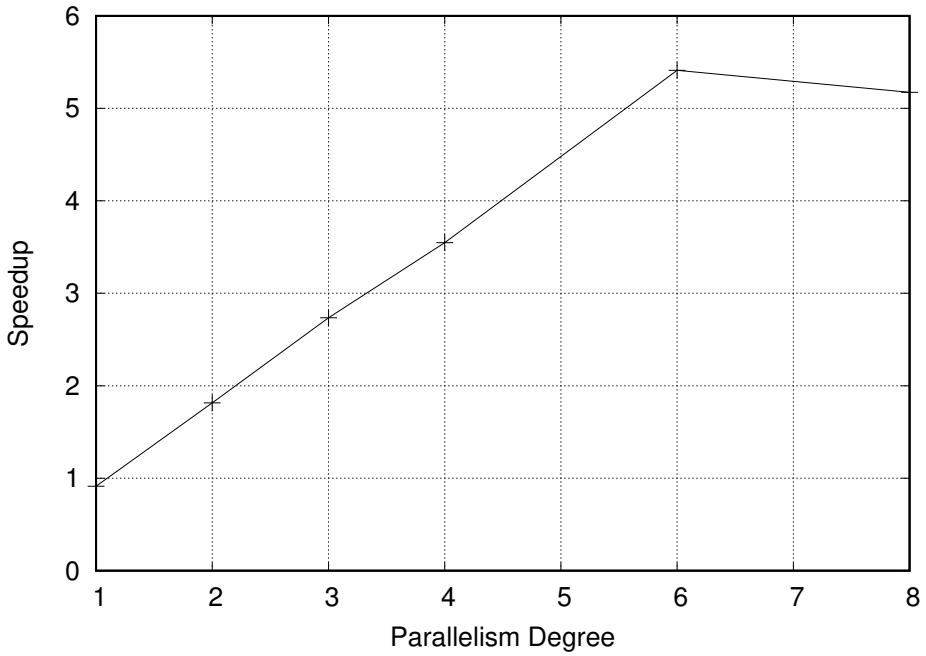


FIGURE 7.14: Absolute speedup of SOP on the four-node Paradigm cluster.

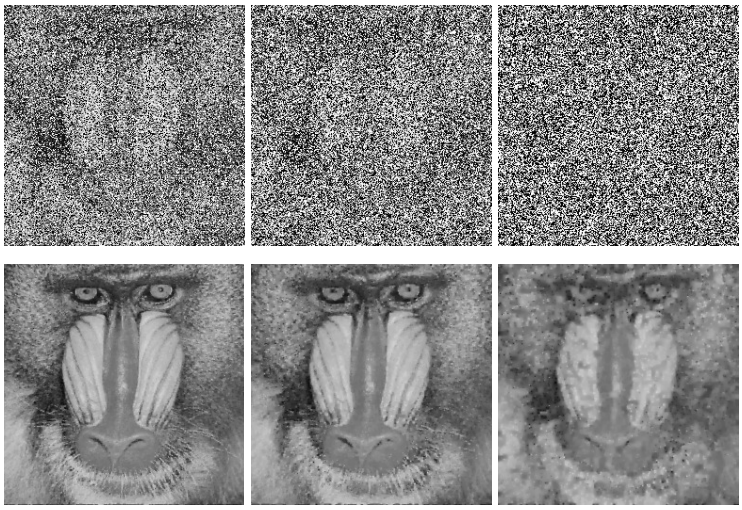


FIGURE 7.15: Restoration filter quality: (first row, left to right) baboon image affected by 50%, 70% and 90% of salt-and-pepper noise; (second row) restoration results (from Aldinucci *et al.* [21]).

Still for this benchmark, we also reported the absolute performance (Fig. 7.10), since it is a crucial aspect when considering the *real-time* nature of video processing. The implemented filter is based on adaptive noise detection and edge-preserving restoration, known to yield high-quality results (cf. Fig. 7.15) but at the price of high computational costs. In this context, the throughput sustained by our GAM implementation is nearly 100 fps (frames per second) in the worst case (i.e., 720p resolution with 70% noisy rate), largely above the usual threshold for real-time processing (viz., 25 fps).

Finally, we remark that the reported maximum throughput is identical on both platforms, since the maximum available parallelism degree for the restoration phase is $n \times m$ (i.e., the number of worker nodes times the number of GPUs per node), which is 8 in both cases.

High-Frequency Stock Option Pricing

Both relative and absolute speedup exhibits the same linear trend as we observed for the previous benchmark, at least as long as the total amount of available parallelism on the cluster nodes is able to sustain the parallelism requested by the number of parallel executors (i.e., the number of farm processors, referred as n). We observe this phenomenon on Paradigm, where the speedup (Figs. 7.12 and 7.14) is linear up to $n = 6$, whereas it drops with $n = 8$. Conversely, the speedup keeps growing linearly on OCCAM within the considered range for n (Figs. 7.11 and 7.13), due to the greater amount of available parallelism.

As for the absolute maximum throughput measured within the considered deployments, the GAM implementation of SOP is able to process about 500000 and 100000 stock options per second, on the OCCAM and Paradigm platform, respectively.

Summary

In this chapter, we provided a preliminary evaluation of the proposed GAM stack, from the performance viewpoint, by executing two non-toy benchmarks over three different configurations of small clusters. We incidentally showed how the proposed GAM stack supports both heterogeneous HPC platforms (by running a benchmark on two cluster-of-GPUs configurations) and various network fabrics (by selecting a different networking hardware for each considered cluster).

Chapter 8

Conclusions

In this thesis, we proposed a constructive approach for providing support for distributed platforms to mainstream C++ programming, with a focus on large-scale HPC platforms. To this aim, we designed a *stack* (Fig. 1.1) of three programming models, each coupled with a C++ API and its implementation. To the best of our knowledge, each stack layer represents a standalone research contribution, as we discuss in the following.

In the bottom layer (Ch. 3), we proposed GAM (Global Asynchronous Memory), a parallel programming model combining features of both the *shared-memory* and *message-passing* models, thus overcoming the traditional dichotomy between these two paradigms. Instead of a full-fledged memory interface (i.e., load/store), we proposed a memory model with a stricter semantics, in which each memory location is either *public* or *private*, and so it can only be accessed in a *single-assignment* or an *exclusive* manner, respectively. This eliminates read-write conflicts, allowing to achieve *sequential consistency* without any coherence support underneath, thus avoiding the major limiting factor for scalability in formerly proposed DSM models. Still along the line of supporting HPC platforms, we implemented GAM on top of *libfabric*, a library for network programming focused on large-scale systems, providing seamless deployment on a wide variety of modern networking standards (e.g., InfiniBand, Intel Omni-Path). In the context of parallel programming models, GAM is the first attempt at providing sequential consistency for the distributed-memory system model, without relying on any hardware or software coherence.

In the second layer (Ch. 4), we proposed *smart GAM pointers* as an extension of C++ smart pointers for referencing GAM memory locations. Smart GAM pointers provide automatic management of dynamic GAM memory, thus guaranteeing the absence of both *memory leaks* and *dangling pointers*, at the level of the whole distributed system at hand. We matched the distinction between public and private GAM locations with that between C++ shared and unique pointers, resulting in public and private pointer APIs, respectively. From a programming perspective, the mechanism of casting smart GAM pointers to plain C++ pointers (and vice versa) allows to target distributed platforms by means of existing C++ code, with minimal programming effort. Smart GAM pointers are the first attempt at expanding automatic memory management to the whole address space in a distributed-memory context.

In the third layer (Ch. 5), we proposed GAM nets, a parallel programming model inspired by FastFlow [17]. Programming in GAM means designing applications as *streaming networks* of processors, whose interactions are based on exchanging pointers. As demonstrated by several works in the

literature, this model allows to implement efficient RTSs for a wide variety of higher-level programming models, from simple data-parallel constructs (e.g., parallel for) to fully fledged DSLs for Big Data analytics. GAM nets is the first API targeting stream-parallel programming over distributed-memory platforms, oriented to support the seamless porting of shared-memory code, through simple pointer casting.

Finally, we provided a preliminary evaluation of the proposed GAM stack, in terms of both expressiveness (Sect. 7.1) and performance (Sect. 7.2). According to the reported results, the proposed GAM API and the associated parallel programming model (i.e., GAM nets) enables the low-effort porting of shared-memory code, from a broad range of application domains; at the same time, the stacked implementation exhibits minimal performance overhead and does not interfere with scalability for the considered benchmarks. Incidentally, we showed the flexibility obtained by implementing our GAM stack on top of libfabric, by performing the experiments on three different network fabrics, namely Ethernet, InfiniBand, and A3Cube RONNIEE (cf. Fig. 2.4).

Future Work

As we discussed in Sect. 6.1, GAM nets could be exploited to realize a library of containers with data-parallel transformations, on the same line as the Nvidia Thrust [112] library.

In the same context, on top of the proposed GAM stack, we envision a deeper integration with modern C++ concepts for parallel programming, for instance, by implementing GAM-based iterators, regarding GAM as an *execution policy* for the ongoing effort about the so-called parallel STL.

As we discussed in Sect. 6.2, GAM could play a central role in the context of developing the RTS for task-based frameworks, targeting distributed platforms.

Moreover, we plan to export the GAM model to a broader class of parallel platforms, in addition to HPC clusters. In particular, the proposed "consistency without coherence" approach could be applied for enabling sequential consistency on those platforms that provide little or no hardware coherence, and also for yielding better performance on cache-coherent systems, putting less stress on the coherence protocol.

Finally, at the very bottom of the proposed GAM stack, we plan to investigate the possibility of exploiting the RMA facilities provided nowadays by a number of networking standards. For instance, the actual memory to which GAM locations are mapped could be allocated, on each executor, from memory exported regions, and made accessible to other executors by means of RMA primitives. In this setting, a remote load would amount to a simple call of a hardware primitive, rather than require a cooperative operation, as in the current implementation (cf. Fig. 3.6).

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, May 2011.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernàndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[5] M. Aldinucci. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters*, 13(3):449–460, Sept. 2003.

[6] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino. The Open Computing Cluster for Advanced data Manipulation (OCCAM). In *The 22nd International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, San Francisco, USA, Oct. 2016.

[7] M. Aldinucci, C. Calcagno, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, M. Torquati, and A. Troina. On designing multicore-aware simulators for systems biology endowed with on-line statistics. *BioMed Research International*, 2014.

[8] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Targeting distributed systems in FastFlow. In *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, volume 7640 of *LNCS*, pages 47–56. Springer, 2013.

[9] M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, M. Torquati, and A. Troina. On parallelizing on-line statistics for stochastic biological simulations. In *Euro-Par 2011 Workshops, Proc. of the 2st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*, volume 7156 of *LNCS*, pages 3–12, Bordeaux, France, 2012. Springer.

[10] M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, M. Torquati, and A. Troina. On designing multicore-aware simulators for biological

systems. In Y. Cotronis, M. Danelutto, and G. A. Papadopoulos, editors, *Proc. of Intl. Euromicro PDP 2011: Parallel Distributed and network-based Processing*, pages 318–325, Ayia Napa, Cyprus, Feb. 2011. IEEE.

[11] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[12] M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, Oct. 2007.

[13] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016.

[14] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, G. Peretti Pezzi, and M. Torquati. The loop-of-stencil-reduce paradigm. In *Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara)*, pages 172–177, Helsinki, Finland, Aug. 2015. IEEE.

[15] M. Aldinucci, M. Danelutto, G. Giaccherini, M. Torquati, and M. Vanneschi. Towards a distributed scalable data service for the grid. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 73–80, Germany, Dec. 2006. John von Neumann Institute for Computing.

[16] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Rhodes Island, Greece, Aug. 2012. Springer.

[17] 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. Wiley, 2017.

[18] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[19] M. Aldinucci, M. Drocco, G. Peretti Pezzi, C. Misale, F. Tordini, and M. Torquati. Exercising high-level parallel programming on streams: a systems biology use case. In *Proc. of the 2014 IEEE 34th Intl. Conference on Distributed Computing Systems Workshops (ICDCS)*, Madrid, Spain, 2014. IEEE.

[20] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with fastflow. In M. Danelutto, T. Gross, and

J. Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, Feb. 2010. IEEE.

[21] M. Aldinucci, G. Peretti Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Applications*, 29(4):461–472, 2015.

[22] M. Aldinucci, G. Peretti Pezzi, M. Drocco, F. Tordini, P. Kilpatrick, and M. Torquati. Parallel video denoising on heterogeneous platforms. In *Proc. of Intl. Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*, 2014.

[23] M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, and S. Palazzo. A parallel edge preserving algorithm for salt and pepper image denoising. In K. Djemal, M. Deriche, W. Puech, and O. N. Ucan, editors, *Proc. of 2nd Intl. Conference on Image Processing Theory Tools and Applications (IPTA)*, pages 97–102, Istambul, Turkey, Oct. 2012. IEEE.

[24] M. Aldinucci, F. Tordini, M. Drocco, M. Torquati, and M. Coppo. Parallel stochastic simulators in system biology: the evolution of the species. In *Proc. of Intl. Euromicro PDP 2013: Parallel Distributed and network-based Processing*, Belfast, Nothern Ireland, U.K., Feb. 2013. IEEE.

[25] M. Aldinucci and M. Torquati. *FastFlow website*, 2009. `http://mc-fastflow.sourceforge.net/`.

[26] M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato. Fastflow: Combining pattern-level abstraction and efficiency in GPGPUs. In *GPU Technology Conference (GTC 2014)*, San Jose, CA, USA, Mar. 2014.

[27] M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato. An overview of fastflow: Combining pattern-level abstraction and efficiency in GPGPUs. In *GPU Technology Conference (GTC 2014)*, San Jose, CA, USA, Mar. 2014.

[28] M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, C. Misale, C. Calcagno, and M. Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, 15(5):798–813, 2014.

[29] J. D. andy Martin Sandrieser andy Siegfried Benkner. Ocr-vx - an alternative implementation of the open community runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, November 2015*, November 2015.

[30] C. Augonnet, O. Aumage, N. Furmento, S. Thibault, and R. Namyst. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. Rapport de recherche RR-8538, INRIA, May 2014.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore

Architectures. In *Proceedings of the 15th International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, Aug. 2009. Springer.

[32] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[33] Beam. Apache Beam website. https://beam.apache.org/.

[34] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault. High-Level Support for Pipeline Parallelism on Many-Core Architectures. In *Europar - International European Conference on Parallel and Distributed Computing - 2012*, Rhodes Island, Grèce, Aug. 2012.

[35] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[36] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 172–183, Dec 2016.

[37] H.-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[38] H.-J. Boehm and S. 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.

[39] Boost. Boost Serialization documentation webpage. http://www.boost.org/doc/libs/1_63_0/libs/serialization/doc/serialization.html.

[40] Boost. *Boost.MPI website*, September 2017 (last accessed). http://www.boost.org/doc/libs/1_65_0/doc/html/mpi.html.

[41] S. Brookes. Deconstructing CCS and CSP. *MFPS16, April*, 2000.

[42] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with OmpSs. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[43] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[44] C. Calcagno, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, and A. Troina. Modelling spatial interactions in the arbuscular mycorrhizal symbiosis using the calculus of wrapped compartments. In

I. Petre and E. P. de Vink, editors, *Proc. of Third International Workshop on Computational Models for Cell Processes (CompMod)*, volume 67 of *EPTCS*, pages 3–18, Aachen, Germany, Sept. 2011.

[45] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, pages 52–60, April 2004.

[46] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. 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.

[47] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. 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.

[48] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[49] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Munster skeleton library Muesli — a comprehensive overview. In *ERCIS Working paper*, number 7. ERCIS – European Research Center for Information Systems, 2009.

[50] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

[51] M. Cole. A skeletal approach to exploitation of parallelism. In *Proc. of CONPAR 88*, British Computer Society Workshop Series, pages 667–675. Cambridge University Press, 1989.

[52] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[53] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[54] M. Cole. *Skeletal Parallelism home page*, 2009. http://homepages.inf.ed.ac.uk/mic/Skeletons/.

[55] M. Coppo, F. Damiani, M. Drocco, E. Grassi, M. Guether, and A. Troina. Modelling ammonium transporters in arbuscular mycorrhiza symbiosis. *Transactions on Computational Systems Biology (TCS)*, 6575(13):85–109, 2011.

[56] M. Coppo, F. Damiani, M. Drocco, E. Grassi, E. Sciacca, S. Spinella, and A. Troina. Hybrid calculus of wrapped compartments. In G. Ciobanu and M. Koutny, editors, *Proc. of 4th Workshop on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC)*, volume 40 of *EPTCS*, pages 102–120, Jena, Germany, Aug. 2010.

[57] M. Coppo, F. Damiani, M. Drocco, E. Grassi, E. Sciacca, S. Spinella, and A. Troina. Simulation techniques for the calculus of wrapped compartments. *Theoretical Computer Science*, 431:75–95, 2012.

[58] M. Coppo, F. Damiani, M. Drocco, E. Grassi, and A. Troina. Stochastic calculus of wrapped compartments. In A. D. Pierro and G. Norman, editors, *Proc. of the 8th Workshop on Quantitative Aspects of Programming Languages (QAPL)*, volume 28 of *EPTCS*, pages 82–98, Paphos, Cyprus, Mar. 2010.

[59] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Compututer Systems*, 8(1-3):205–220, 1992.

[60] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Proc. of 6th Intl. Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 1175–1184, Munich, Germany, Aug. 2000. Springer.

[61] M. Danelutto and M. Torquati. Loop parallelism: a new skeleton perspective on data parallel patterns. In M. Aldinucci, D. D'Agostino, and P. Kilpatrick, editors, *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE.

[62] 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*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.

[63] F. Darema. *The SPMD Model: Past, Present and Future*, pages 1–1. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[64] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In *Proc. of Parallel Architectures and Langauges Europe (PARLE'93)*, volume 694 of *LNCS*, pages 146–160, Munich, Germany, June 1993. Springer.

[65] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 19–28, New York, NY, USA, 1995. ACM.

[66] T. De Matteis and G. 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 '16, pages 13:1–13:12, New York, NY, USA, 2016. ACM.

[67] T. De Matteis and G. Mencagli. Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. *International Journal of Parallel Programming*, 45(2):382–401, 2017.

[68] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Usenix OSDI '04*, pages 137–150, Dec. 2004.

[69] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, May 2017.

[70] J. Dokulil and S. Benkner. Towards high-level parallel patterns in OpenCL. In *2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 199–204, Dec 2014.

[71] J. Dokulil, M. Sandrieser, and S. Benkner. Implementing the open community runtime for shared-memory and distributed-memory systems. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 364–368, Feb 2016.

[72] M. F. Dolz, D. del Rio Astorga, J. Fernández, J. D. García, F. García-Carballeira, M. Danelutto, and M. Torquati. Enabling semantics to improve detection of data races and misuses of lock-free data structures. *Concurrency and Computation: Practice and Experience*, 29(15), 2017.

[73] M. Drocco, M. Aldinucci, and M. Torquati. A dynamic memory allocator for heterogeneous platforms. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, Fiuggi, Italy, 2014. HiPEAC.

[74] M. Drocco, C. Misale, and M. Aldinucci. A cluster-as-accelerator approach for SPMD-free data parallelism. In *Proc. of Intl. Euromicro PDP 2016: Parallel Distributed and network-based Processing*, pages 350–353, Crete, Greece, 2016. IEEE.

[75] M. Drocco, C. Misale, G. Peretti Pezzi, F. Tordini, and M. Aldinucci. Memory-optimised parallel processing of Hi-C data. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, pages 1–8. IEEE, Mar. 2015.

[76] M. Drocco, C. Misale, G. Tremblay, and M. Aldinucci. A formal semantics for data analytics pipelines. Technical report, Computer Science Department, University of Torino, May 2017.

[77] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[78] J. Enmyren and C. 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.

[79] Flink. Apache Flink website. https://flink.apache.org/.

[80] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[81] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[82] I. Gartner. Gartner hype cycle. http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp.

[83] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, Nov. 2010.

[84] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, fifth edition, 2011.

[85] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[86] P. Inaudi. Progettazione e sviluppo di un provider libfabric per la rete ad alte prestazioni ronniee/a3cube. Master's thesis, Computer Science Department, University of Torino, 2015.

[87] Intel. *Intel ® C++ Intrinsics Reference*, 2010.

[88] Intel. *Intel ® AVX-512 instructions*, 2013. https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[89] Intel Corp. *Threading Building Blocks*, 2011.

[90] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[91] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[92] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[93] Khronos Compute Working Group. *OpenCL*, Nov. 2009. http://www.khronos.org/opencl/.

[94] Khronos Group. *SYCL website*, September 2017 (last accessed). https://www.khronos.org/sycl.

[95] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[96] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[97] Libfabric. Libfabric OpenFabrics website. http://libfabric.org/.

[98] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[99] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[100] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proc. of the 1st Inter. conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[101] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The open community runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2016.

[102] Memcached. Memcached website. https://memcached.org/.

[103] I. Merelli, F. Tordini, M. Drocco, M. Aldinucci, P. Liò, and L. Milanesi. Integrating multi-omic features exploiting Chromosome Conformation Capture data. *Frontiers in Genetics*, 6(40), 2015.

[104] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[105] C. Misale. *PiCo: A Domain-Specific Language for Data Analytics Pipelines*. PhD thesis, Computer Science Department, University of Torino, May 2017.

[106] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. In *Proc. of HLPP2016: Intl. Workshop on High-Level Parallel Programming*, pages 1–19, Muenster, Germany, July 2016. arXiv.org.

[107] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017.

[108] C. Misale, M. Drocco, G. Tremblay, and M. Aldinucci. Pico: a novel approach to stream data analytics. In *Euro-Par 2017 Workshops - Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-Dasp)*, Santiago de Compostela, Spain, 2017. (Accepted).

[109] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. *CoRR*, abs/1504.00788, 2015.

[110] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[111] NVIDIA Corp. *CUDA website*, June 2017 (last accessed). https://developer.nvidia.com/cuda-zone.

[112] NVIDIA Corp. *Thrust website*, June 2017 (last accessed). https://developer.nvidia.com/thrust.

[113] S. Oaks and H. Wong. *Java Threads*. Nutshell handbooks. O'Reilly Media, 2004.

[114] OpenFabrics Interfaces Working Group. *Libfabric OpenFabrics Programmer's Manual*, 2017. https://github.com/ofiwg/ofi-guide/blob/master/OFIGuide.md.

[115] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[116] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9:143–161, 2001.

[117] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[118] J. H. Rutgers. Programming models for many-core architectures: a co-design approach. 2014.

[119] L. M. Sanchez, J. Fernandez, R. Sotomayor, S. Escolar, and J. D. Garcia. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing*, 31(3):139–161, 8 2013.

[120] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[121] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[122] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In *Proceedings of the 12th International Conference on Parallel Computing Technologies*, pages 258–272, St. Petersburg, Russia, Oct. 2013.

[123] F. Tordini, M. Drocco, I. Merelli, L. Milanesi, P. Liò, and M. Aldinucci. NuChart-II: a graph-based approach for the analysis and interpretation of Hi-C data. In C. D. Serio, P. Liò, A. Nonis, and R. Tagliaferri, editors, *Computational Intelligence Methods for Bioinformatics and Biostatistics - 11th International Meeting, CIBB 2014, Cambridge, UK, June 26-28, 2014, Revised Selected Papers*, volume 8623 of *LNCS*, pages 298–311, Cambridge, UK, 2015. Springer.

[124] F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci. Parallel exploration of the nuclear chromosome conformation with NuChart-II. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*. IEEE, Mar. 2015.

[125] F. Tordini, M. Drocco, C. Misale, L. Milanesi, P. Liò, I. Merelli, M. Torquati, and M. Aldinucci. NuChart-II: the road to a fast and scalable tool for Hi-C data analysis. *International Journal of High Performance Computing Applications (IJHPCA)*, pages 1–16, 2016.

[126] 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. *Journal of Supercomputing*, 2017. To appear.

[127] UniTo-INFN. Occam Supercomputer website. http://c3s.unito.it/index.php/super-computer.

[128] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX.

[129] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 423–438, New York, NY, USA, 2013. ACM.

[130] ZeroMQ. *website*, 2012. http://www.zeromq.org/.

[131] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.

Glossary

API Application Programming Interface

BSP Bulk Synchronous Parallel

CCS Calculus of Communicating Systems

CSP Communicating Sequential Processes

DAG Direct Acyclic Graph

DRF Data Race Free

DSL Domain-Specific Language

DSM Distributed Shared Memory

DV Data-Value

FIFO First-In First-Out

FLOPS Floating Point Operations Per Second

FPGA Field Programmable Gate Array

GAM Global Asynchronous Memory

GAS Global Address Space

GOS Global Object Space

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HPC High Performance Computing

IaaS Infrastructure-as-a-Service

ISA Instruction Set Architecture

LTS Labeled Transition System

MIMD Multiple Instruction Multiple Data

MPMC Multiple Producer Multiple Consumer

MPSC Multiple Producer Single Consumer

MSI Modified Shared Invalid

NGS Next-generation Sequencing

NUMA Non-Uniform Memory Access

PE Processing Element

PGAS Partitioned Global Address Space

PN Process Network

RMA Remote Memory Access

RTS Run-Time System

SC Sequential Consistency

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SMP Symmetric Multiprocessor

SNL Software Network Layer

SPMC Single Producer Multiple Consumer

SPMD Single Program Multiple Data

SPSC Single Producer Single Consumer

STL Standard Template Library

SWMR Single Writer Multiple Reader

TSO Total Store Order

UMA Uniform Memory Access

VPU Vector Processing Unit