Mémoire

présenté à

L'Université de Bordeaux

École Doctorale de Mathématiques et Informatique

par Olivier AUMAGE

Chargé de recherche INRIA

Équipe-projet commune STORM
Équipe SATANAS
INRIA – LABRI

en vue de l'obtention du diplôme d'

Habilitation à Diriger des Recherches

Spécialité : Informatique

Instruments of Productivity for High Performance Computing

Soutenance le : 14 décembre 2020

Après avis de :

M.	Siegfried	BENKNER	Professeur à l'Université de Vienne, Autriche	Rapporteur
M.	Franck	CAPPELLO	Directeur de Recherche, Argonne National Lab., États-Unis	Rapporteur
Mme	Elisabeth	LARSSON	Professeure à l'Université d'Uppsala, Suède	Rapporteuse

---

Devant la commission d'examen formée de :

M.	Luc	GIRAUD	Directeur de recherche, INRIA Bordeaux – Sud-Ouest	Président
M.	Frédéric	DESPREZ	Directeur de recherche, INRIA Grenoble – Rhône-Alpes	Examinateur
M.	Christoph	KESSLER	Professeur à l'Université de Linköping, Suède	Examinateur
M.	Enrique	QUINTANA–ORTÍ	Professeur à l'Université Politechnique de Valence, Espagne	Examinateur
M.	Siegfried	BENKNER	Professeur à l'Université de Vienne, Autriche	Rapporteur
M.	Franck	CAPPELLO	Directeur de Recherche, Argonne National Lab., États-Unis	Rapporteur
Mme	Elisabeth	LARSSON	Professeure à l'Université d'Uppsala, Suède	Rapporteuse
M.	Denis	BARTHOU	Professeur à Bordeaux INP	Directeur

Melancholy . . . A Researcher's Instrument of Productivity ?

In memorie

Black ink marker drawing on paper, Olivier AUMAGE, April 2020.
After drawings by A. DÜRER & M. C. ESCHER, and painting by F. DEL COSSA.

Acknowledgements

I would like to thank Elisabeth LARSSON, Siegfried BENKNER and Franck CAPPELLO for their work in reviewing my manuscript, Christoph KESSLER, Enrique QUINTANA-ORTÍ and Frédéric DESPREZ for their participation to my *habilitation* thesis defense jury and Luc GIRAUD for presiding the jury.

I am very much grateful to Denis BARTHOU for his support during the preparation of this thesis. I would specially like to thank the members of Team STORM, our team assistant Sabrina DUTHIL, and the great support staff of INRIA. Finally, I warmly thank all the former and current colleagues, for their numerous suggestions, contributions and helping hands all along these two decades of research in High Performance Computing.

Abstract High performance computing (HPC) is now well established as the cornerstone for building and conducting software simulations in numerous scientific and industrial fields. The hardware complexity of supercomputers is steadily increasing, however, to deliver ever improved computing performance, causing the complexity of HPC application development to increase as well. As a result, the need for tools and methodologies to compensate the development complexity by reducing the costs it induces —that is, the need for instruments to improve productivity in HPC development— has never been so pressing. This manuscript builds on the experience I have gathered while being involved in the design of several of such instruments of productivity for HPC, over the course of my academic career in computer science research, to study the technical choices made, the lessons learnt, and to discuss upcoming challenges and priorities.

Keywords Parallelism; Cluster; Runtime System; Language; Interoperability; Composition; Tuning.

Résumé Le calcul intensif (HPC) est désormais bien établi comme la pierre d'angle pour construire et mener des simulations logicielles dans d'innombrables domaines scientifiques et industriels. Cependant, la complexité des supercalculateurs croît de façon soutenue, pour fournir des performances de calcul sans cesse améliorées, provoquant par là même la complexification du développement d'applications HPC. En conséquence, le besoin d'outils et de méthodologies pour compenser la complexité du développement en réduisant les coûts qu'elle induit — c'est-à-dire le besoin d'instruments de productivité pour le développement HPC — n'a jamais été aussi pressant. Ce manuscrit s'appuie sur l'expérience que j'ai acquise durant ma carrière académique dans la recherche en informatique à travers la conception de plusieurs de ces instruments de productivité pour le HPC, afin d'étudier les choix techniques effectués, les enseignements, et de discuter des défis et priorités à venir.

Mots-clés Parallélisme ; Cluster ; Support d'exécution ; Langage ; Interopérabilité ; Composition ; Adaptation.

Contents

Introduction

Chapter 1

Introduction

1.1 High Performance Computing

Software simulation is now commonplace in scientific research and development, from academic to industrial institutions. Indeed, simulations can complement, prepare, and in some cases even entirely supplant real life experiments —with substantial economical benefits as the most immediate advantage— while enabling new studies for which real experiments would be impractical. Simulations typically involve phenomena being formalized, and discretized into problems to be solved on computers. The accuracy of a simulation therefore depends on the discretization coarseness: the more refined the discretization, the more accurate a simulation, the larger the size of the simulation problem, and the larger the amount of computing resources needed to solve it, in terms of computing units, memory space, and network bandwidth.

In practice, many simulation problems of interest to the academic and industrial scientists do require vast amounts of computing resources to produce relevant results in an acceptable time frame. The purpose of the *High Performance Computing* (HPC) branch of computer science is to design supercomputer hardware, software tools and programming techniques to meet this objective of *solving large relevant scientific problems in an acceptable time frame*. The qualifiers and quantifiers —"large", "relevant", "acceptable time frame"— are intentionally left uninstantiated here, since they highly depend on the application domain and on specific constraints: for instance, the acceptable time frame for running a climate evolution model simulation can be much larger than the acceptable time frame for a weather forecast which needs to complete before the forecasted date; yet, both are archetypal HPC applications.

Hardware vendors for HPC are in a constant innovation process to offer increasingly powerful supercomputers. However, this innovation process comes at the price of increasing hardware complexity: since the reaching of the so-called "frequency wall" in the mid-2000 years, vendors cannot anymore increase the processor clock's frequency significantly further —and certainly not by the factors commonly seen during the decades before that— due to the difficulty in extracting the resulting extra heat dissipated by processors; instead, they have to design techniques to do more per clock cycle. Such techniques typically include instruction level parallelism with out-of-order, superscalar execution and single instruction/multiple data (SIMD) instruction sets, chip level parallelism, with hardware multithreading, multicore, and multiprocessor architectures. They also include improvements in the ecosystem surrounding processors, such as dedicated memory controllers, cache hierarchies to reduce processor stalls,

and smarter peripherals enabling the processor to offload some specific tasks, such as network packet processing, graphical scene generation, or even numerical kernel computation on accelerator boards. Many of these techniques originated long before the frequency wall was reached, however they are now the key drivers of computing power growth. It is therefore of critical importance for HPC applications to take advantage of them.

1.2 Performance Portability

The fast pace of HPC hardware evolution, and the rate at which its complexity increases, make it difficult for software development to keep-up, though. For instance, the family of SIMD instruction sets in INTEL processors has been enhanced every two years, on average, since the initial MMX technology was introduced in 1997 inside the INTEL PENTIUM processor. The TESLA microarchitecture introduced in 2006, the first microarchitecture of NVIDIA graphical processors to enable general purpose computations, has been successively superseded by six microarchitectures since then, each with new capabilities and features. The landscape of networks for HPC also has seen dramatical evolutions in the last two decades, with promising technologies becoming obsolescent, such as the QUADRICS QSNET or MYRICOM MYRINET networks in the 2000s, and the emerging of new technologies such as the Aries interconnect from CRAY or more recently the BXI interconnect from ATOS/BULL, while the INFINIBAND family of interconnects has instead seen a surprising and rare longevity. To cope with this diversity and this continual hardware evolution, applications programmers require a property from HPC programming software called *performance portability*, that is, the ability for the resulting program to run efficiently on a variety of hardware technologies with only moderate adaptation redevelopment costs. Performance portability has been the main guiding thread of my research work over the last twenty years, through my involvement in the design of several *communication runtime systems* and *computing runtime systems*.

Compilers have long been the main workhorse in enforcing performance portability. Depending on the language compiled, they manage to achieve a high level of abstraction for the programmer, while taking advantage of the large corpus of compiler technique research to generate heavily optimized programs tailored to the target architecture. This contribution of compilers to performance portability is commonly complemented by specialized numerical libraries implementing a set of topically related numerical kernels, thoroughly optimized for some hardware. Well known examples include libraries of linear algebra building block routines (for instance the BLAS —Basic Linear Algebra Subroutines— libraries), or Fast Fourier Transform routines (e.g. the `libfftw`). Hardware vendors typically supply variants of these libraries optimized for their processors. Yet, the complexity of modern HPC hardware makes it impractical to implement performance portability exclusively from a static point of view, relying on the application compile time and the library link time to make decisions over the lifespan of the application execution. These difficulties come from multiple different sources, such as the variability of computer resources' performance, the combinatorial explosion caused by the number of available computing resources/memory-hierarchies/devices, some inherent irregularity in the application algorithm, or event some system noise. In facing them, static-only approaches suffer from their lack of adaptivity throughout the program execution, that is, their lack of adaptivity *at run-time*: this is naturally where runtime systems come into play.

Runtime systems are software layers linked with an application and/or with numerical

libraries, whose responsibility is to drive the processing of work requests expressed by the application or libraries onto some hardware resources. This description is generic enough to encompass multiples large classes of runtime systems, such as computing runtime systems driving computational work requests (application routines, numerical library kernels), runtime systems for networking driving communication work requests (e.g. data transfers between nodes), and others. In this respect, a runtime system resides on the critical path of the work requests flow, which may be a prominent cause of the programmers' wariness when considering the use of runtime systems: having an extra software layer to traverse seems counterproductive on the path to performance, let alone performance portability; yet, a runtime system is essential in several ways. First, it provides programmers with stable, abstract programming interfaces enabling an application code to be written in a device independent manner. This is indeed the most desirable property at the source of the success of OPENGL and DIRECTX, two runtime systems —well known in the domain of computer graphics— allowing graphic application programmers and computer game programmers to design their codes in a largely hardware oblivious way. Then, it can help with the tedious process of hardware discovery and let programmers defer the selection of hardware resources at run-time, when a better informed decision can be taken. Second, a runtime system provides applications with adaptivity, by detecting hardware resources and capabilities (or the lack of them), by calibrating with their current level of performance measured, and by detecting load imbalance through continuous accounting and observation of resource usage. And last but not least, a runtime system is at the right place in a software stack to apply optimization strategies on the global flow of work requests, in order to drive better overall performances.

My research work on runtime systems has successively been dealing with network runtime systems, computing runtime systems, and more recently within distributed computing runtime systems, on multiple aspects such as programming model abstraction, resource mapping and execution model, adaptivity and work request optimization. Based on this experience, the first main part of this manuscript, page 11, converses about the topic of runtime system design for the High Performance Computing domain, with an objective of performance portability.

1.3 Programmability Enhancement

In order to be effective in delivering performance, runtime systems must be fed with a flow of work requests that contains opportunities for such performance delivery. For instance, a computing runtime system will not be able to take advantage of a processor's multiple cores if it is fed with a sequential flow of computing requests that does not contain indications that some of these computations can safely by executed concurrently. Thus firstly, the application code has to be designed to generate such a flow of requests, then, these requests have to leave enough freedom of movement to the runtime system to be able to take non trivial decisions in realizing them; and finally, the requests must come with appropriate semantics annotations for the runtime system to ponder whether some optimizing steps are safe to be performed or not: even while relying on runtime systems, the process of writing applicative codes for HPC inherently requires a specific expertise often beyond the skills of a scientific simulation programmer in a given application domain, because acquiring such an expertise would be too time consuming or too expensive to obtain, and would be a distraction from the scientific domain in focus. Higher level approaches to writing applicative codes are therefore necessary

in the HPC software ecosystem, and the challenge is designing them in a way that conciliates the requirements for a wide target audience, and the requirements for collecting, preserving and funnelling performance opportunities down to the runtime system.

The second main part of this manuscript, page 61, discusses three directions I explored as part of my research work on *programmability enhancement* for HPC applications. A first direction explores the use of the OPENMP parallel language and the KSTAR compiler to generate code for driving a computing runtime system with a level of performance comparable to the direct use of the runtime system's own native programming interface. A second direction explores going one step further, by building a composite HPC application from multiple pre-existing parts, with the aim of preventing the performance opportunities generated in each part to cancel each others in the composite application. A third direction explores the idea of physically separating the work of writing the scientific domain simulation code on the one side, and implementing optimization strategies on the other side; that is, the idea of separating concerns between domain specific scientific programmers and specialists of HPC code optimisation.

1.4 Performance Tuning

The runtime systems discussed above, as well as the popular compilers, all make some assumptions on the applicative code layout and properties that implicitly define the respective scopes for which their optimization capabilities are most effective. Some applicative codes unfortunately fall outside these scopes. The third and last main part of this manuscript, page 101, deals precisely with this situation. It presents several research works I have been involved in, the common purpose of which has been to define performance tuning approaches to bring back some applicative codes within the optimizing scope of general purpose compilers and runtime systems.

A first work presented in this part explores the matter of task granularity tuning, in the case of applicative codes generating excessively thin computing work requests that would be too expensive to process individually. This work investigates the definition of systematic preprocessing operators to aggregate several tasks together, and make them heavy enough for the runtime system action to be profitable despite its operating overhead. A second work explores the case of applicative codes that fall outside the scope of automatic vectorizing compilers. From an analysis of the compiled binary code and from memory access execution traces, it aims at pinpointing vectorization hindrances, and at suggesting data layout transformations and code transformations to the programmer for bringing the application back within vectorizing compilers' reach. A third work explores the case of the family of "polar" error correction code algorithms, for whom a number of algorithm-specific optimizations are known, but entirely outside the scope of a general purpose compiler. This work studies the design of a source code generator, generic for the whole "polar" codes family, but specific to it, to perform the particular optimization work upstream, before letting the general purpose compiler do its own job.

1.5 Overall Objectives and Manuscript Organisation

From these three tightly related points of view of running, programming and tuning applicative codes for High performance Computing, the objectives of this manuscript are to

offer an overview of the research works I have conducted and taken part in, a synthesis of the lessons learnt, and a discussion of the directions and perspectives I find important from there. It is by no means exhaustive, and does not pretend to be, but it should be an useful guide for the scientific programmer feeling lonely on the verge of porting his/her application to the HPC world, in discovering the instruments and techniques that are already there to lend a helpful hand at each step of the process.

During my research work, I have on several times followed a pattern of designing a runtime software engine, then of building one or more programming interfaces on top of it to make it accessible to a wider range of programmers, and finally of integrating tunable knobs and means to let users go further and widen the range of targeted applications. Thus, instead of a purely chronological organization or an architecture oriented top-down or bottom-up layout, I have chosen to follow this same progression to organize this manuscript. Hence, Part I first starts with discussing the design of HPC runtime system engines and architectures with a primary focus on performance portability. Part II then investigates approaches to enhancing the programmability of HPC platforms. Finally, Part III enquires into the performance tuning of some HPC applicative codes that do not readily fit into the scope of general purpose HPC instruments.

Part I

Approaches to Performance Portability

Table of Contents

On June 25, 2018, the US Department of Energy's Oak Ridge National Laboratory (ORNL) presented their SUMMIT supercomputer [156], ranking 1st on the TOP500 list of worldwide fastest supercomputers announced on the same day. Beyond its 200 petaflops theoretical peak performance, and 122.3 petaflops achieved on the LINPACK [68] benchmark, this supercomputer presents several interesting characteristics. As virtually every supercomputer nowadays, it is laid out as a cluster of computer nodes interconnected with a high performance network, here of MELLANOX INFINIBAND (IB) EDR 100 GB/s technology. Each node is heterogeneous, meaning it hosts more than a single kind of computing processor: A SUMMIT node contains two 22-core IBM POWER9 processors and six accelerators NVIDIA VOLTA V100. The cores of the IBM POWER9 processors come with 4 hardware threads each, totalling 176 CPU threads per nodes. Each node is also heterogeneous in terms of memory, equipped with DDR4 memory, high bandwidth memory (both addressable by CPUs and GPUs), and non-volatile memory useable as burst buffer or extended memory. In France, the new supercomputer JOLIOT-CURIE from GENCI, the entity in charge of supervising national-level supercomputing facilities, also exhibits heterogeneity, but of a different kind than SUMMIT. JOLIOT-CURIE is an ATOS/BULL SEQUANA supercomputer organised as two partitions, a thin-node partition of general purpose INTEL SKYLAKE processors, and a many-core partition equipped with INTEL 68-core KNL processors.

The architectural and technical differences between these two machines make it difficult to design an application that works efficiently on each of them, even though they opened for production roughly at the same time. On a computer node equipped with six high-throughput VOLTA accelerators, one major challenge is to be able to feed each accelerator with data at a sufficient rate for it to work at full speed, while avoiding interferences and competitions with other accelerators accessing memory on the same node. On many-core KNL nodes instead, the challenge is more to find sufficient parallelism of the right grain to give work to all the cores. In both cases, however, a common additional challenge is to have many parallel entities on the same node share access to the networking hardware in an efficient and highly reactive way.

Moreover, the hardware is itself evolving quickly. The high bandwidth memory (HBM) and its competitor MCDRAM installed in INTEL KNL processors only started to be widely available a few years ago —the INTEL KNL was launched in 2016—. Non volatile memory is only beginning to be available in supercomputers. On the processor side, the IBM POWER9 processor doubles the number of cores from the previous generation. The NVIDIA VOLTA V100 has more than 5 000 cores, with double-precision and half-precision floating point computing capabilities, while the first TESLA accelerator introduced by NVIDIA ten years ago only had 128 cores and only supported single precision computing. The INTEL KNL is a standalone processor, while the previous generation of INTEL many-cores were supplied as discrete extension board running alongside a general purpose CPU.

On the network side, both SUMMIT and the thin-node partition of JOLIOT-CURIE use similar MELLANOX INFINIBAND EDR networks. The MELLANOX IB EDR network implements one of the most recent evolution of the INFINIBAND technology standard, originating from the INFINIBAND Trade Organization founded two decades ago in 1999. In contrast, the many-core partition of JOLIOT-CURIE is interconnected by the BXI interconnect [64] from ATOS/BULL, a brand new interconnect in the landscape of high performance networks. Thus, here also, hardware is heterogeneous: while both networks have some nominal characteristics in common, such as high bandwidth and low latency capabilities, an application written on top of INFINIBAND `ibverbs` API would not be readily portable on the BXI network, and conversely,

an application using BXI's triggered operations capabilities —that is, operations offloaded onto the BXI network interfaces cards and processed automatically in the background— would likely be cumbersome to port on the INFINIBAND `ibverbs` API.

This diversity in HPC hardware is even more obvious in the whole TOP500 list. On November 2018, the list named about ten network technology classes (e.g. ETHERNET, INFINIBAND, CRAY ARIES and GEMINI, INTEL OMNI-PATH, etc.) and about ten more listed as just "custom" or "proprietary" interconnects. In terms of processors, the TOP500 lists a vast majority of supercomputers based on INTEL X86-64 architectures; however, a significant part of the actual computing power available in that list is brought by a more diverse class of architectures including the POWER9 of SUMMIT cited above or the SUNWAY processor of the TAIHULIGHT machine.

Programmer Dilemma Confronted with these hardware evolution trends, a programmer faces a dilemma. On the one side, the requirement of portable programming to keep development and maintenance costs under control encourages a conservative approach, to stick with established programming techniques and legacy programming interfaces, such as provided by the operating system and the C language standard library (`libc`) designed for long-term portability and stability. However, such APIs are not well suited to HPC hardware: for instance, the cost of going through the *socket* network stack is way too expensive compared to the sub-microsecond latency of high performance networks. On the other side, the incentive of using the "bleeding edge" performance enhancing and convenient features of modern hardware. For instance, some modern network boards are able to offload tasks such as message tag matching, handshake protocol operations (e.g. rendez-vous), and even encryption; accelerator devices are able to offload expensive computing kernels, leaving the main CPUs to focus on general purpose tasks. However, such features are not usually available through legacy APIs, and may require non-portable hardware- or vendor-specific developments.

Runtime Systems: Cornerstones to Performance Portability In November 2017, the European Technology Platform for HPC body (ETP4HPC) published its third Strategic Research Agenda (SRA) [32], the multi-annual roadmap for European HPC Technology, in which it stressed the need for a closer integration of runtime systems in the programming environments and toolchains used to develop high performance computing applications, to support programming models able to target modern heterogeneous platforms and to help in their adoption. Along a very similar line, the Exascale Computing Project (ECP) in the United States, involving the U.S. Department of Energy and other bodies, published the ECP Software Technology Capability Assessment Report [92] on July 1st, 2018, in which it stated, under the "Programming Models & Runtimes" technical area entry: "*We are also developing a diverse collection of products that further address next generation node architectures, to improve realized performance, ease of expression and performance portability.*". There is therefore a growing trend in strategy definition bodies to see runtime systems as an important component for performance portability, itself a key factor to make applications quickly run efficiently on new computing platforms.

In the following chapters, we discuss the design and evolution of runtime systems for communication, for intra-node parallel computations, and for inter-node distributed computations, in light with our work and experiments in these domains.

Chapter 2

Performance Portability of Communications

Since their emergence as computing platforms in the '90s, computer clusters, or *networks of workstations* [8, 152] as they were also called by then, have been quickly adopted to become the dominant form factor for HPC supercomputers. The fundamental hardware technology that made this evolution possible is the technology of high performance networks. High performance networks, such as the Myrinet, Quadrics QSNet, SCI, or Infiniband networks, for instance in those early days, and such as (still) Infiniband, Intel Omni-Path Architecture, ATOS/Bull BXI or Cray Gemini and Aries networks nowadays, all come with distinct programming interfaces, and even distinct programming paradigms sometimes, thus raising both portability of performance and mere portability issues for applications. We present in this chapter the work we have conducted on network runtime systems MADELEINE [16] and NEWMADELEINE [17, 39] to address these issues.

2.1 Context: High Performance Networks Technologies

High performance networks interconnect the computing nodes of clusters with high bandwidth, low latency links, so as to let applications *potentially* benefit as much as possible from the cluster-wide aggregated computing power. However, traditional networking software stacks such as the "Socket" API associated with the TCP/IP protocols would have a significantly negative impact on the performance actually observed by applications: system calls to the operating system would harm latency, intermediate copies would harm bandwidth, and algorithms such as congestion control and fragmentation/reassembly management would harm both. Thus, in lieu of relying on legacy communication stacks, high performance networks are instead programmed through dedicated interfaces, some standardized, some proprietary. These interfaces largely share the same constraints, brought by the necessity to bring both high-bandwidth and low-latency hardware network access to user application processes, within the technical boundaries of the modern operating systems' protection models.

2.1.1 User-Level Networking

Indeed, a key property of modern operating systems is to isolate user processes from hardware components. While this property is desirable for security reasons, it is a potential issue for high performance networking, since requiring systems calls or memory copies on the critical path to drive a network interface card (NIC) would harm the latency and bandwidth effectively achievable by applications, in comparison to the theoretical hardware capabilities. To overcome this issue, high performance NICs support being directly mapped into user process memory spaces. The mapping operation itself still involves system calls, but at initialization time, rather than on the critical path. Once the card is mapped in a user process memory space, that process can directly interact with the card, without involving the operating system anymore. This direct interacting mode is called *OS-bypass*.

2.1.2 Memory Management

In reality, and with the exceptions of network interface cards integrating their own memory management unit (MMU) —such as the now defunct Elan/Quadrics technology— only the Programmed Input/Output (PIO) operating scheme is available in a *fully* OS-bypass mode, meaning that the CPU has to be actively moving bytes from the process memory to the NIC for sending messages, and likewise in the opposite direction for receiving. While this can be acceptable for short messages, this becomes an issue for transferring long messages without the possibility to overlap these transfer times with computations.

NICs do have the capability to perform Direct Memory Access (DMA) data transfers, to discharge the CPU from the tasks of moving bytes. However, there is a technical limitation to use them from within user processes: operating systems virtualise the memory space of user processes, which means that the memory addresses manipulated by user processes are *virtual* addresses, only meaningful inside these processes. In contrast, NICs only deal with *physical* addresses, as do other hardware devices. Thus, in the general case, a NIC would not be able to make sense of a process-supplied pointer expressed as a virtual address (again, with the exception of network interface cards equipped with a MMU). In consequence, an user process must instead obtain the corresponding physical address, that is, the physical *mapping*, of a virtual address pointer, in order to pass it to a NIC to designate the source or destination buffer of a data transfer.

Moreover, that mapping may change over time, as operating systems take advantage of the virtual memory model to transparently —from the point of view of the user process— move virtual memory pages across the physical memory, or even to disk, when page swapping is triggered. In that situation, a virtual-to-physical mapping obtained previously suddenly becomes obsolete without the user process being notified of the change, because of the virtual memory management inherent transparency. Such an event would be disastrous for subsequent communication attempts by the process. To avoid this, it is necessary to prevent the virtual memory management of the operating system to alter the mapping of virtual memory pages during communications, with an operation called *memory pinning*.

As a matter of fact, obtaining and pinning the mapping between a virtual and a physical memory page address necessitates a system call, since the page table keeping track of physical/virtual address mappings is managed by the operating system. Thus, operating system calls cannot be entirely avoided when using DMA mechanisms from a user process. Since the cost of these operating system calls is roughly on the same order of magnitude, at the least,

than the minimal transfer time of the interconnect, it cannot be neglected. In consequences, a variety of techniques are employed to mitigate the resulting overhead, by reusing pinned memory allocations, either though a pool of pre-pinned buffers, or by managing a cache to keep track of pinned memory areas and try to spread the pinning cost over multiple data transfer requests.

2.1.3 Paradigms

The way for an end-user code to interact with NICs depends on the interconnect technology. Some vendors supply specific, proprietary APIs, some other vendors employ standardized software stacks such as, notably, the OPENFABRICS ALLIANCE stack commonly found on top of INFINIBAND interconnects. Yet, the communication paradigms implemented by such NIC APIs all are variants of two fundamental paradigms. These paradigms are named the *one-sided* communication paradigm and the *two-sided* communication paradigm, respectively. The two-sided communication paradigm typically defines some sending and receiving routines, and requires both sides of a communication two actively take part in the processing of a communication request, one by issuing a send operation, and the other one by issuing a matching receive operation. In the one-sided communication paradigm instead, only one side of a communication, the *initiator*, takes an active part in the communication request, while the other one, the *target*, remains passive; one-sided operations initiated by the sending side are often called *put*, *remote write* or RDMA [1]-Write, while those initiated by the receive side are called *get*, *remote read* or *RDMA-Read*. Some NIC APIs are strongly or exclusively based on one of these two paradigms (INFINIBAND, for instance is mainly based on one-sided communications, the Myrinet MX API [98], instead, favored two-sided communications), while some other APIs combine both (such as CCI [11], for instance).

2.2 Implementing Performance Portability on High Performance Networks

The consequence of the characteristics of high performance interconnect technologies, briefly summarized in the paragraphs above, is that many implementation choices have to be made in targeting them in end-user programs. These choices involve trade-offs, such as selecting among various communication strategies, various memory management strategies, selecting the most suitable paradigm, all this while keeping in mind that a successful combination of design choices made in targeting some hardware on a given platform may perform inefficiently or even be entirely unavailable on another platform.

The MADELEINE [16] and NEWMADELEINE [17, 39] communication libraries are two communication runtime systems we successively designed to address the issue of providing applicative code with performance portability in programming high performance networks. The MADELEINE library was our first attempt at adopting a two-layer architecture to implement portable communication programming through abstraction. It enabled simultaneously offering a dynamic selection of communication methods taking into account the communication requests' parameters on the one side, and the characteristics of the selected interconnect device on the other side, to make an efficient mapping of these requests on low level hardware-specific communication routines. The NEWMADELEINE library was an entire redesign aiming

1. Remote Direct Memory Access

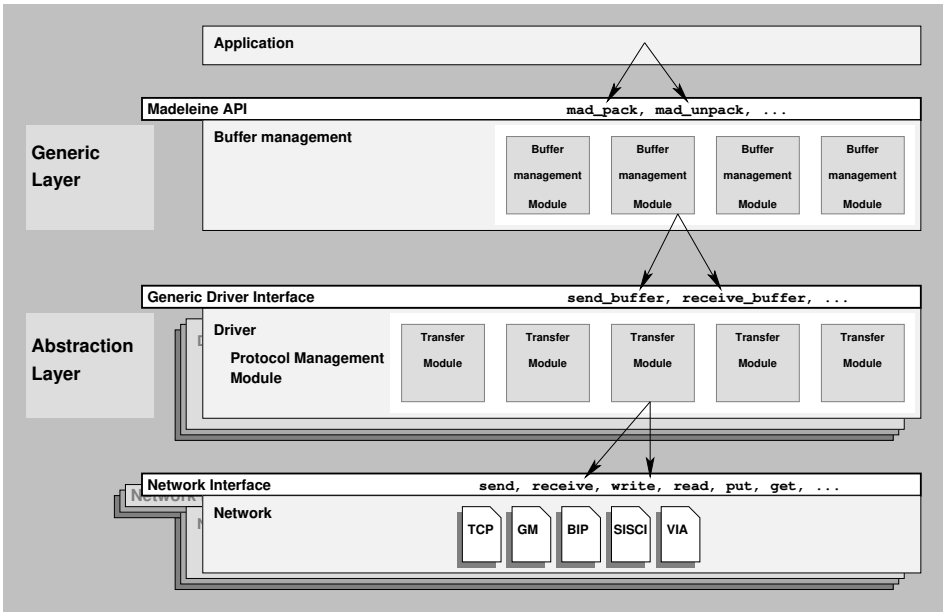


Figure 2.1 – Architecture of the MADELEINE communication library.

at rationalizing the communication optimization process, while extending its scope to support non-deterministic optimization schemes by implementing a programmable communication request *scheduling engine*.

2.3 The MADELEINE Communication Library

The MADELEINE [16] communication library was designed around the end of the '90s decade. In the HPC domain, one of the major evolution of that decade was the emergence of clusters of workstations, mentioned above, made using *off-the-shelf* hardware components, as a challenging alternative to traditional custom-made supercomputers. One of the driver of that evolution was the strong progress being made at the time on high performance interconnects, making it possible to link multiple nodes efficiently in a cost effective way. In return, the emergence and fast spreading of clusters of workstations as HPC platforms drove a blooming interconnect market, with many interconnect vendors popping up, pushing many diverse, competing, and naturally incompatible technologies. In this context, the purpose of the MADELEINE communication library was to offer a performance portability layer on top of these interconnect technologies to the PM^2 [121] framework, a distributed parallel programming framework based on the remote procedure call (RPC) paradigm.

The architecture of the MADELEINE library is represented in Picture 2.1. It follows the typical runtime system organization with a hardware-dependent low-level layer made of drivers, and a portable user-facing API. The API exposes a set of routines to build messages —on the sending side— and extract messages —on the receiving side— in an incremental manner, and to specify semantics annotations on each message fragment. The sequence of message building calls on the sending side is required to match the sequence of message extraction calls on the receive side, in terms of fragment size, order and semantics annotations.

The *generic driver interface* exposes a set of abstract routines to send/receive single buffers

and groups buffers, as well as a device-dependent decision function. Between the library API and the *generic driver interface* of the *abstraction layer*, a generic buffer management layer is responsible for collecting data from the applicative code and building network messages along the most efficient approach for the selected network driver, using calls to the corresponding driver's decision function to guide the buffer management and optimized message building process.

The message building process in the upper, generic layer is organized as a static and deterministic decision tree. The device-dependent decision functions are required to give deterministic decisions from identical input parameters on a message fragment, on a per-connection basis. Thus, the sending side of a point-to-point network connection is guaranteed to take the same sequence of decisions than the receive side. This deterministic process enables sending data with little to no payload overhead, depending on the underlying network characteristics: only the flow-control related overhead, such as rendez-vous handshake or credit-based accounting, is needed if the underlying network API necessitates them.

The static decision tree and requirements for a deterministic decision process, while enabling a very low overhead on a per-message basis, are limiting the scope of applicable optimizations, however. In particular, they hinder the ability of the network runtime system to use its cross-cutting position in the communication stack to optimize communication requests in a global manner.

Moreover, the session management model of the MADELEINE library is a *closed* session group model. All the application processes on the participating nodes are launched at startup time, and the connectivity between them is also established at startup, to last throughout the whole session time. While this model is usually sufficient for many usages on small to moderate size computing sessions —indeed, this is the model most commonly available and used with MPI— this may cause scalability issues, due to the possibly prohibitive startup time on large configurations; this prevents modular, dynamic sessions where new connections are established or closed according to evolving needs, and this may also limit the possibility to subsequently integrate fault tolerance support to the library due to the impossibility to close failed connections and establish new ones without relaunching the whole session.

The NEWMADELEINE library, presented next, is an entirely re-designed communication library, offering an open, dynamic session management model, and conceived with the aim to emphasize its runtime system role as an application-wide communication request scheduler over a narrower per-message optimization scope.

2.4 The NEWMADELEINE Communication Library

The main design philosophy of the NEWMADELEINE [17, 39] communication library is to decouple the user-facing API, used to build communication requests, from the actual processing of such requests. In contrast with the MADELEINE library, where a call to the API could trigger a chain of calls all the way down to the network interface, NEWMADELEINE's API calls only append requests to request lists in a non-blocking manner, while the requests themselves are then processed in the background at the network pace.

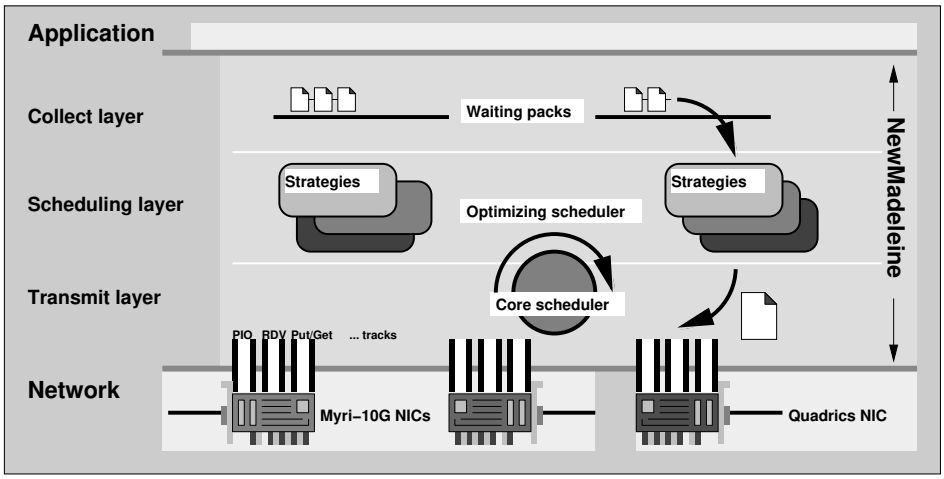


Figure 2.2 – Architecture of the NEWMADELEINE communication library.

2.4.1 Architecture

The architecture of the NEWMADELEINE library is shown in Figure 2.2. It is organized in three layers. The *Collect layer* is responsible for collecting communication requests from the applicative code; the *Scheduling layer* is responsible for optimizing the flow of communication requests; the *Transmit Layer* is responsible for passing communication requests down to the underlying network interface. The three layers each are modular and extensible themselves: the *Collect layer* offers multiple flavors of user-facing APIs, including a MPI-compatible flavor [17, 168]; the *Scheduling layer* features programmable scheduling strategies; the *Transmit layer* naturally implements multiple network drivers. Only the *Transmit layer* contains device-dependent code.

The decoupling of the *Collect layer* from the *Transmit layer* enables an adaptive processing of communication requests. The management of device's activity/idleness periods and the selection of *polling* versus *blocking* device interactions and monitoring are done asynchronously in the *transmit layer*, in cooperation with the PIOMAN service [168]. While the *Transmit layer* is busy, the *Collect* layer accumulates communication requests in a list of posted requests. When the *Transmit layer* is about to become idle, it queries the *Scheduling layer* for more work. The *Scheduling layer* uses the communication request list as an optimization window to decide about the communication operation(s) to perform next.

2.4.2 Communication Requests Scheduling

The *Scheduling layer* employs programmable *scheduling strategies* to decide about communication operations. These strategies can use a number of possible actions:

— Merge several requests into a single request, using memory copies;
— Aggregate requests into a single logical request, using gather/scatter mechanisms;
— Reorder requests, to prioritize some over others, or to enable additional merge/aggregation opportunities;
— Split requests, to fit maximum transfer unit requirements;
— Select among multiple transfer modes to execute a request, such as two-sided send/receive primitives or one-sided remote DMA operations, for instance, when available on

the selected device;
— Load balance requests over multiple devices or multiple device ports, by keeping track of the amount of work assigned to each networking resource and of the progress of that work;
— Process a communication request over multiple network links (e.g. *multirail* data transfer), by sending distinct fragments of a message over each link concurrently.

The Scheduling layer is helped in this decision process by quantitative and qualitative informations from network device drivers about feature availability (gather/scatter, for instance) and device capacity (multiplexing limits, maximum transfer units, for instance). It can also use some information collected from sampling the actual platform's network(s) performance under the various available usage modes, to determine which transfer mode or network link is more efficient for a given kind of communication request.

To take the full advantage of this panel of network devices capabilities, it is necessary to remove the constraint of processing communication requests in-order, one at a time, per connection: Instead of using two identical decision trees on the send and receive side, as was the case with the MADELEINE library, the NEWMADELEINE library moves the decision process entirely on the sending side. This means that NEWMADELEINE removes the requirement for deterministic request processing sequences. As a consequence, the library is responsible for adding headers, when necessary, to let the receive side identify incoming operations and message fragments.

However, the rationale of the NEWMADELEINE design is that the impact of this network payload overhead may generally be mitigated on modern network interfaces, by using tagging facilities, inline header data, and/or gather/scatter transfer methods to avoid the cost of extra low level network requests. In return, removing the requirement for deterministic processing enables a whole range of opportunistic optimizations, where the request scheduler can take advantage of multiple independent communication requests being posted in a short time-frame to optimize them as a whole.

Such a global optimization capability is especially important on many-core network nodes, where the large number of cores may hit a bottleneck in accessing network devices; thanks to opportunistic optimizations, the bottleneck can instead be turned into an advantage, in a transparent manner for the applicative code.

The global optimization capability is also important in terms of modular interoperability. Modern HPC applications commonly make use of specialized, numerical libraries that may access the network for their own purpose, concurrently with the application core and possibly with other libraries as well. The scheduling layer of NEWMADELEINE may not only avoid conflicts in network resource access between these pieces of software, but also optimize their respective communication patterns as a whole.

2.4.3 Programming interfaces

There is no consensus in the community about a unique programming interface for network programming that would fit every usage: one-sided communication primitives (e.g. `put` / `get`, `RDMA_Read` / `RDMA_Write`), two-sided communication primitives (e.g. `send` / `receive`), with or without incrementally building messages, with or without triggering of a programmer supplied receive handler routine, etc. Thus, the *Collect layer*, in charge of collecting communication requests from applicative codes and libraries built on top of NEWMADELEINE, does not impose a unique programming interface. It instead comes with a

variety of predefined programming interfaces.

Thanks to the abstraction provided by the NEWMADELEINE architecture, all the interfaces implemented in the Collect layer do not require any specific support in the other layers, and the set of interfaces can therefore easily be extended with additional ones as needed; in return all these interfaces benefit from the optimizations implemented in the Scheduling layer. At the time of writing, the set of interfaces shipped with NEWMADELEINE includes the `sendrecv` interface, the `pack` interface implementing incremental message building with send/receive semantics, the `rpc` interface implementing a *Remote Procedure Call* distributed programming model, to name the main ones. Moreover, it offers a comprehensive MPI [17, 168] programming interface, which enables the vast corpus of existing MPI applications and codes to readily run on top of NEWMADELEINE and benefit from request scheduling optimizations as well.

2.5 Discussion

The NEWMADELEINE library constitutes a versatile communication processing engine to mutualize HPC network management services in higher level distributed computing environments, such as explored in Section 4. However, many issues remain open and new ones emerge due to the evolution of HPC supercomputing platforms.

2.5.1 One-sided Communication Primitives

While *one-sided communication primitives* (see Section 2.1.3) have been available in low-level APIs of interconnects such as INFINIBAND for more than twenty years now, taking a real advantage of them all the way up to the applicative code remains challenging. The limited adoption of MPI's one-sided routines in real applications is an indication of this difficult issue. The fundamental reason of the difficulty resides in the fact that taking advantage of one-sided communication primitives requires the *initiator* process to have prior knowledge of the internal state of the *target* process in order to launch a one-sided communication request: since the target is passive, the initiator must know that the remote memory being read from with a `get` or being written to with a `put` can safely be accessed from an algorithmic point of view; that is, a `get` will obtain a valid, complete, up-to-date piece of data from a correct (e.g. allocated and accessible) memory area; a `put` will target a valid, authorized memory area and will not unexpectedly overwrite data that has not yet been processed. In some cases, this prior state knowledge of the target can been known easily over a long period of time (relatively to the applicative code makespan), such as accessing a constant, read-only database with `get` operations. In some other cases, the state knowledge requirements can be relaxed to some extent, such as observing some memory area with `gets` for visualization purpose (depending on whether the visualization code can accommodate un-synchronized data reads); such a possibility may also exist in rare cases for `put` operations if the applicative code algorithm may support un-synchronized writes or successive writes without flow control. In the general case of HPC applicative codes, however, any prior knowledge about a process state detained by another process is limited.

Inside a communication library such as NEWMADELEINE, the situation is even worse, since the request processing engine only gets a narrow view of the communication scheme unfolding, without indications of its insetting in the application code algorithm as a whole.

Non-deterministic optimizations consisting in replacing a `send`/`recv` pair of primitives with a `put` or a `get` primitive are possible only by primarily performing a handshake exchange between the initiator and the target, so that the target does not trigger an unnecessary `recv` or `send`, and guarantees that the memory area targeted is ready. This limits the scope of one-sided primitive optimization initiatives that the library can take on its own, and the benefit of these optimization initiatives due to the necessary handshake: A net gain is only possible if such a handshake can be shared between multiple communication requests.

Thus, to maximize the performance benefit of one-sided primitives, work is needed at multiple levels:

— At the applicative code level, the algorithm must be designed so as to maximize the number of one-sided updates requests achievable in a row without synchronization with a peer process. Such a requirement may involve trade-offs such as allocating more communication buffers to enable more concurrent in-flight communication requests, thus trading increased memory usage for a lower execution time.

— Also at the applicative code algorithmic level, the communication pattern should be designed so as to maximize the sharing of synchronizations operations over multiple communication requests. This is an approach notably explored with the notion of *shared notifications* [5].

— A cooperation must be implemented between the applicative code and the networking library, since even though the applicative code may have little knowledge of remote process states, it still has more of that knowledge than the library. Such a cooperation could be done, for instance, by using one-sided requests directly and explicitly at the library API level when possible, since converting the requests afterwards within the communication engine is difficult, and by proactively hinting about one-sided compatible communication sequences on the application side, while exploiting such hints on the library side.

As the downside, however, the resulting applicative code may end-up being less abstract, thus less performance portable, and perhaps less portable also.

Partitioned Global Address Spaces (PGAS) programming environments [30, 176, 101] can help simplifying the work of applicative code programmers in leveraging one-sided communication primitives, by providing such programmers with a convenient, human-friendly abstraction. PGAS environments offer a concept of virtual, global memory space spanning multiple nodes, which each participating process can access. However, PGAS environments by themselves do not eliminate the issue of portability of performance but only shift it: the applicative code must take into account whether pieces of data being accessed are local or remote, to avoid frequent expensive remote accesses. Moreover, the issue remains of designing the applicative code algorithms such as to favor sharing state synchronization operations between two processes over the longest possible sequences of transfers.

2.5.2 Session Management

One of the design decision in the NEWMADELEINE library and also in the NETIBIS composable communication stack presented in Section 6.1 was to adopt an open session model, enabling the establishment and termination of connections at any time, instead of establishing all connections at startup in a closed session manner. Such an open session design comes with several benefits. It enables sessions to evolve dynamically, thus to adapt networking resource allocation to actual needs at various steps in the applicative code lifetime.

It can be used as a basis for implementing fault tolerance mechanisms by making it possible to drop a failing connection without bringing down the whole session, and even to establish new connections to a backup process. It can be used for fine grained networking resource multiplexing and communication flows isolation for supporting multiple software modules or multiple threads to communicate concurrently, along a similar idea followed by the concept proposal of MPI *endpoints* [65].

Open sessions may also help with the scalability of the startup sequence by gradually establishing connectivity instead of connecting all participating processes at once. In this objective of startup and internal networking resource scalability, avoiding the creation of large amounts of connections can also be realized, in a complementary manner, by letting applications build sparse connectivities, while supplying forwarding mechanisms at the communication library level. The principle in that case is therefore to trade increased latencies between certain pairs of infrequently communicating processes for reduced connection resource consumption. An extreme example is the case of *stencil*-like classes of distributed applications, where processes mainly communicate with a small number of neighbors, and for which establishing a full connectivity is often overkill. The latest revision of the MPI specification [118] supports some of these capabilities, in terms of connectivity management, with the notions of *process groups* and *intercommunicators*; yet, MPI routines such as `MPI_Comm_spawn` or `MPI_Comm_connect/MPI_Comm_accept` that enable spawning additional processes or connecting with more existing processes are collective operations, thus not that flexible. To address these limitations, some efforts are underway at the MPI FORUM to simplify and possibly automatize part of this work through the notion of MPI *sessions* [93, 94] in cooperation with a runtime system or an external session manager such as PMIX [48].

2.5.3 Requests Processing Clustering

Scalability in communication management is indeed affected by the density of connectivity, but it may also be affected at the level of the interconnected nodes themselves. As the hardware parallelism increases in processors (see Section 3.1.1), the number of processing units concurrently communicating may constitute a possible bottleneck at the communication library level. Techniques such as the MPI *fine points* [86], where multiple applicative threads jointly contribute pieces of a message, or *shared notifications* [5] where a thread manages multiple incoming messages' notifications on behalf of other threads, may reduce some of this bottleneck risk, but imply a level of multi-thread coupling that may not necessarily fit the applicative algorithm.

An alternative is the clustering of requests processing in multiple instances, each instance supervising communications for a subset of computing units. This techniques is often used informally by end-users, when launching sessions with multiple processes per physical nodes, such as the frequently used "one process per processor socket" strategy. Yet, this informal approach introduces an unnecessary level of data partitioning within physical nodes. The communication library may instead help by offering communication clustering capabilities, and possibly leveraging multiple hardware network interface cards, when available, in doing so.

2.5.4 Hardware Acceleration

In terms of exploiting the hardware potential, making use of advanced offloading capabilities offered by modern network interface cards in communication runtime systems would likely provide substantial performance gains. For instance, a last generation CONNECTX-6 INFINIBAND NIC from MELLANOX offers offloading mechanisms for operations such as message tag matching, as well as "rendez-vous" handshake management for long messages, two critical, expensive and ubiquitous operations in HPC communications. Some NICs are even user programmable, embedding a general purpose CPU (for instance an ARM processor), and in some cases also a FPGA circuit. In a similar way to the domain of graphic computing and GPUs, some effort is likely needed on the design of communication runtime systems to help take advantage of some of these hardware capabilities in a portable, and importantly in a performance portable manner.

2.6 Conclusion on Performance Portability of Communications

In this chapter, we investigated performance portability from the point of view of runtime systems for communication. Based on the learnings from the design of the MADELEINE communication runtime —from my own PhD work— and the NEWMADELEINE communication runtime —of which I developed the initial design and implementation— we saw how the interplay of the networking interface cards with the protection mechanisms of modern operating system kernels heavily constrained the design of high performance communication interfaces. Despite this consideration, the diversity of hardware-specific and vendor-specific communication interfaces makes targeting them a non-portable task. Thus, we showed how communication runtime systems can build abstract, hardware-independent application programming interfaces on top of that diversity. Moreover, thanks to a static decision tree in the case of MADELEINE and a programmable communication requests scheduling engine in the case of NEWMADELEINE, we showed how a communication runtime system can still take advantage of hardware specificities to optimize data transfers in a performance portable way. From this now fairly stable core, we then discussed some important directions to explore or not explore from there.

Next chapter will now look into the other main family of runtimes used in HPC, the runtime systems dedicated to the management of computations.

Chapter 3

Performance Portability of Parallel Computations

Processors have continuously been characterized by their diversity in terms of instruction sets, reflecting experiments with several philosophies over time: the complex instruction set computers (CISC) emphasizing rich functionality, the reduced instruction set computers (RISC) emphasizing a streamlined instruction set design separating load/stores from computation instructions, or the very large instruction word approach (VLIW) integrating parallelism at the instruction level, are well known examples of this diversity. Moreover, processors also naturally exhibit diversity in terms of capabilities and power. Such a diversity has mostly been hidden so far, however, by compilers on the one side, taking care of generating the appropriate instructions, and by the operating system on the other side, in charge of mapping computing work requests on available processing units. Thus, the need for runtime systems dedicated to computation may not necessarily appear that obvious at first sight.

Yet, compilers act in a statical manner, before the execution of the applicative code for classical compilers, or immediately before the first execution of each routine for "just-in-time" compilers. Their ability to subsume runtime system requirements is therefore incomplete: hardware used for compiling may not necessarily be the same as hardware used for execution, and decision making may only use information known at compile time, though routine versioning techniques may allow for some decisions to be deferred.

Operating systems provide for hardware abstraction as well as for some relatively coarse resource management and arbitration. However, the focus of a general purpose operating system is on providing processes of potentially multiple applications —and also multiple users for multi-user systems— with a fair, equal and controlled access to hardware, along a *no policy* philosophy. Other, "non general-purpose" OSes exist also: real-time operating systems are dedicated to offering guarantees that some time-critical software completes its tasks within some predefined deadline, thus a different purpose than HPC applications in general; lightweight operating system kernels such as McKernel [83] instead are kernels dedicated to HPC usage and designed to offer only the bare-level set of system services needed to run some applicative process in single user mode, in order to eliminate any performance jitter caused by unnecessary multi-user, fairness-oriented mechanisms. However, such lightweight kernels offer little in terms of fine-grain computing resource abstraction and management by design.

In contrast, the focus of a computing runtime system is to implement portability of

performance, *within* the process(-es) constituting an application run of a single user, that is, on top of the hardware resources assigned to such process(-es) by the operating system; and to apply some specific policy to do so. Thus, such a runtime system basically *complements* the operating system with portability of performance. After an overview of the relevant hardware characteristics of modern processors, we present two approaches to computing runtime systems that we have been involved in: an approach based on the parallel thread paradigm [38, 37], and an approach based on the parallel task paradigm [13, 3, 2, 116].

3.1 High Performance Processor Technologies

This section presents relevant traits of modern processing units for HPC, driving the design of computing runtime systems. It first deals with general purpose CPUs, with an emphasis on hardware parallelism. It then introduces computational accelerators, noting how they compare and differ from general purpose processors. It also discusses relationships between processors and the memory hierarchy.

3.1.1 General Purpose Processors

A general purpose processor is schematically composed of a control unit in charge of fetching and decoding instructions, and one or more execution units, such as the arithmetic and logic unit or the float point unit, in charge of executing those instructions. Since the time needed to access a piece of data in memory is around two orders of magnitude larger than the duration of a nominal clock cycle duration on modern processors, the processor also includes a hierarchy of memory caches from the larger but slower outermost ones to the smaller but faster innermost ones, in an attempt to reduce the cost of repeatedly accessing the most recently/frequently used pieces data.

Until the first half of the 2000s years, the main driver of performance increase from a generation of processors to the next was the increase of the processor's clock frequency. However, due to the physics of integrated circuits, increasing the clock frequency increases the thermal dissipation of the circuit as well, up to the point where cooling down the circuit to keep it from burning becomes intractable. Thus the clock frequency of processors mostly stagnates since the middle of the 2000s. As a consequence, the vendor strategy to supply new processors with ever increasing performance has now shifted to doing more instructions by units of time, that is, to increase parallelism. It is important to note that parallelism in processors existed long before that; the change here is in bringing it to the forefront among the factors affecting performance. Parallelism in processors, or so-called hardware parallelism, is made available in processors under two orthogonal forms, instruction-level parallelism, and chip-level parallelism.

Instruction-level Parallelism Instruction-level parallelism (ILP) globally designates any capability to execute more than a single instruction of the *same* instruction flow in concurrency. This can be achieved in several, combinable ways.

A first approach to ILP is through pipelining. Processors instructions typically are executed by going through a series of steps. This series varies with the brand and model of processors, but commonly includes steps such as `fetch` (obtain the instruction encoded sequence from memory), `decode` (process the encoded sequence to determine the instruction

opcode and its arguments), `read` (get operands from memory, if needed), `execute` (perform the instruction), `write` (send results to memory if needed). A pipeline basically enables an instruction i_2 to proceed with a pipeline stage immediately after the instruction i_1 that it follows has completed this stage and entered the next one.

A second approach consists in using several execution units, either with distinct purposes such as an integer unit and a floating point unit, or with an identical purpose such as two or more integer units, or even both. Processors using this technique, named *superscalar* processors, attempt to take advantage of the inherent parallelism in a single flow of instructions: if two successive instructions i_1 and i_2 do not depend on each other, that is if none of the operands read by i_2 are written by i_1, their order of execution can be relaxed so as to take advantage of idle execution units, thus potentially benefiting from instruction-level parallelism while reducing pipeline stalls.

A third approach to instruction-level parallelism is to apply the same instruction to a collection of data. This approach called SIMD (Single Instruction Multiple data) applies SIMD variants of arithmetic and logical operators to the elements of special-purpose SIMD registers. These SIMD operators present themselves as optional extensions of CPU instruction sets. In contrast to the first two techniques, *pipelining* and *out-of-order* execution, which are used transparently by the processor, this third technique necessitates that the routines be written using special processor instructions. Depending on the way a SIMD-enabled routine actually makes use of SIMD registers, whether it uses all the registers' elements or only a subset of them, whether shuffle instructions have to be inserted or not, the resulting "SIMD-ized" code may perform better or worse than its non SIMD-enabled variant. Determining whether the appropriate SIMD instruction set extension is available on a given CPU, and if so, whether it is worthwhile to use a specially written SIMD version of the programmer's kernel routine, or to rather keep using the scalar (non-SIMD) variant actually is a better option on the machine used, is an archetypal performance portability issue that a computing runtime system may address.

Chip-level Parallelism Chip-level parallelism designates any hardware support to execute several instruction flows concurrently, within a single computing node. Here also, this can be achieved in multiple, combinable ways, and can also be combined with ILP techniques as well. A first technique basically replicates the logic circuitry inside the control unit to drive multiple instruction flows sharing a set of execution units. This technique, called *hardware multithreading*, itself can be implemented along three different ways: fine-grained multithreading (FMT), coarse-grained multithreading (CMT), or simultaneous multithreading (SMT). In the FMT implementation, each hardware thread executes for one cycle at a time in a round-robin fashion. In the CMT implementation, each hardware thread executes in coarse, round-robin chunks up to a blocking point such as a pipeline stall. So, in both FMT and CMT, the hardware threads are actually interleaved rather than in concurrence. However, the fast thread switching hardware mechanisms in both implementations enable the processor to potentially hide the cost of expensive data accesses in memory of one thread with some computation on another thread, provided the applicative code supplies sufficient parallelism to do so. The SMT implementation brings hardware multithreading one step further, by enabling multiple flows of instructions to concurrently compete for accessing the execution units; provided a sufficient number of execution units is available, the processors can execute instructions from multiple flows simultaneously, hence the name. A second technique to

chip-level parallelism is to replicate the whole processor chip itself in so-called *multiprocessor* architectures, enabling each one of the multiprocessor's CPUs to take part in executing distinct instruction flows. Finally, a third, intermediary approach is to replicate the circuit entity constituted by the control unit and its execution units (and possibly some cache memory) —named the processor core— within the chip, to create a *multicore* processor.

These levels of parallelism can be combined, so a modern high-performance computer can use a multiprocessor architecture, with each individual processor being itself a multicore, possibly with multiple hardware threads per core, each core being itself pipelined, superscalar, and SIMD-enabled. The resulting architecture can thus be rather complex. Moreover, the computing resources may be heterogeneous. Processor architectures such as the BIG.LITTLE one from ARM are multicore architectures in which the cores are not all identical: the "big" cores use a more powerful but less energy efficient electronic circuit design, while the "little" cores follow the opposite strategy. The complexity and diversity of chip-level parallelism bring many performance portability issues, such as deciding which hardware parallelism level to leverage for a given applicative code on a given machine: using two hardware threads of a single core to process two work requests can be beneficiary if these work requests interleave well, especially in the case of simultaneous multithreading, where hardware threads directly compete for execution units; assigning two work requests on two cores of the same processor or two hardware threads of the same core can be beneficiary if these work requests do not interfere in accessing the cache memory spaces they share; assigning a work request to a slow, low energy consuming core instead of a fast, high energy consuming one can be an energy saving option or a loss depending on energy costs of other components in the system such as the memory, for which the energy bill of a longer work request processing can exceed the gain on the computing core energy bill. Resolving these trade-offs is typically among the purposes of a computing runtime system.

3.1.2 Accelerators

When designing processors, hardware designers have to decide how to use the real estate, that is, the surface area of the processor die. Basically, this area has to be organized in terms of subareas dedicated to control, to computation, and to memory caching circuitry. General purpose processors are designed, by definition, to perform well on a wide range of codes: scientific applications, office applications, web browsers, databases, etc. as well as the operating system itself. To do so, hardware designers optimize the use of the die area towards circuitry to "dampen" the complexity and diversity of these codes. Concretely, this means dedicating a substantial fraction of the die to execution control, integrating mechanisms such as advanced branch predictors to reduce the impact of conditional branchings on the instruction pipeline, as well as prefetching mechanisms attempting to bring relevant data from main memory into processors caches in a timely manner, even for complex stridden accesses, again to prevent pipeline stalls. And consequently, besides control, a large part of a general purpose processor die is dedicated to cache memory. The underlying principle of hardware accelerators is to adopt the opposite approach: instead of optimizing the die area usage for general purpose workloads, it is optimized for a very specific class of workloads, by making assumptions about the layout, the control flow and the properties of the kind of codes to be run on them.

The concept of *accelerator* started to gain traction in HPC around the mid-2000s. CLEARSPEED Company in Bristol, UK released the X620 boards in 2006, used in particular in Tokyo

Institute of Technology's TSUBAME supercomputer. The X620 board is equipped with CLEARSPEED's custom processor CSX600, an array of 96 double precision floating-point processing elements running at 210 MHz, thus a large number of processing elements able to perform computations in parallel, but running at a much lower frequency and much higher energy efficiency than general purpose processors, and with only a very small amount of cache-like memory (6 KBytes of SRAM, much less than common CPUs). The company disappeared by the end of the 2000s decade, however. Over the same period, IBM commercialized the CELL Broadband Engine Architecture [58], well known to have been the main processors of SONY's PLAYSTATION 3, but also used in supercomputers such as the ROADRUNNER in Los Alamos National Lab., USA. The CELL BE processor associates a general purpose processor core with a team of "Synergistic Processor" cores acting as processing elements. It also became discontinued without immediate successor, however. Yet, some design aspects of these two processors can be found again in subsequent hardware designs based on processor arrays, called "manycore" processors, such as the INTEL XEON PHI (60-core KNC board, 64..72-core KNL processors), or the INTEL SCC [96] (48 Pentium cores organised in 24 tiles), the MPPA [61] processor from KALRAY (80 processing elements organized in 5 clusters, in the MPPA3/COOLIDGE processor), the SUNWAY SW26010 processor (organised as 4 clusters of 64 Compute Processing elements) found in the SUNWAY TAIHULIGHT supercomputer in Wuxi, China, to name a few. The most common aspect in these hardware designs is the fact that the processor die area is used to favor increased hardware parallelism while using simpler cores with simpler control units and smaller amounts of cache or scratch memory.

Also around the same time as CLEARSPEED's X620 announcement, in 2006, NVIDIA in California released the first graphic boards equipped with their G80 Graphic Processing Unit (GPU), the first GPU featuring a so-called "unified shader architecture". This architecture became significant to HPC in that besides purely graphic-related computations such as triangle meshes processing, it was also able to perform scientific application computations —though only in single precision floating point— with a high degree of parallelism and a high level of performance on some kernels compared to CPUs. The NVIDIA board 8800 GTX of the G80 series, for instance, had 128 "stream processors" (SP) and a peak performance of 350 single precision GFLOPS. SPs are organized in teams called "warps". Each SP in a warp acts as a hardware thread, with the particularity that execution control is shared among all the threads of a warp, and thus the same instruction is dispatched to all of them within the warp. The consequence is that in the case of branching code (e.g. `if/then/else`), some of the threads in the warp can stall, lowering effective parallelism. Such accelerators are therefore optimized for highly parallel and regular (i.e. with few branches) codes.

Due to the assumptions made in accelerators' design, a given applicative code may perform better or worse than on a general purpose processor, depending on the extent to which it fits these assumptions. Moreover, if the accelerator resides on an expansion board, which is still the case for many accelerators, the input data has to be transferred from main memory to the accelerator board, the kernel has to be launched with the appropriate mechanism, and the result has subsequently to be fetched back at some point. Hence, the potential gain induced by the accelerator processor must exceed the cost of launching the processing on the accelerator plus the cost of such data transfers to be worthwhile. This question of worthiness is again a performance portability issue.

3.1.3 Memory Layout and Management

The relationship between processors and memory is naturally tight. Both influence each others' design and characteristics to mitigate the "memory wall" problem [173], due to the gap between the processor clock rate and the memory clock rate. We already encountered examples of such mutual influence when studying chip-level parallelism above (see Section 3.1.1), and hardware multithreading in particular, since one purpose of hardware thread models is an attempt to hide the cost of accessing memory. Memory is itself evolving to try and keep up with processor requirements, such as with the introduction of high-bandwidth memories [133].

Memory Hierarchy The large discrepancy between the processor clock rate and the memory clock rate has led vendors to intercalate memory caches on the data path between processor units and the main memory. Memory caches are fast memory areas where recently accessed data are kept for faster subsequent references. As a consequence of being much faster than the main memory based on dynamic RAM (DRAM), the static RAM (SRAM) used for caches is also much more expensive, hence used only in smaller quantities. High-end HPC processors nowadays commonly intercalate three levels of caches, from the innermost, smallest and fastest L1 cache to the outermost, largest and slowest L3 cache [52]. The L1 cache is in the order of a few KBytes to a few dozens KBytes large. The L3 cache reaches more than 100 MBytes on the IBM POWER9 processor. Some versions of the POWER9 and of the INTEL SKYLAKE processors, for instance, can also support a L4 cache. The L1 cache is usually private to each core. The L2 cache can be private or shared between cores, depending on the architecture. L3 and L4 are usually shared.

Cache management, data eviction and memory consistency enforcement are largely transparent for programmers, who can however give some hints: e.g. data that should be prefetched into some cache level, write operations that should not be cached (called *non-temporal* writes). Some processors optionally support cache partitioning, though some early experiments with the capability seem to suggest that the benefit is limited for HPC workloads [21]. The result of these architecture layouts is an increasingly deeper "memory hierarchy", which has to be taken into account to maximize the ratio of memory accesses hitting the innermost caches, while minimizing caches misses resulting in expensive accesses to the main memory; in contrast, when the notion of memory wall emerged [173] in 1995, HPC processors only used a single level of cache.

In addition to its deepening, on the processor side, the memory hierarchy is also complexifying and deepening on the memory side as well: HBM memory technologies, with their distinct latency/bandwidth profiles from "classical" memory technologies, bring additional options —thus additional choices to be made— about where to put data; also, emerging non-volatile memory technologies, slower than HBM and classical DRAM, but able to preserve data over machine restarts and much faster than usual storage technologies, offer what amounts to one more level in the memory hierarchy. Moreover, effort is also on-going to offer in-memory processing capabilities [105, 107], with some electronic memory modules able to perform arithmetic computations and binary logic operations locally, without involving the CPU.

Non-Uniform Memory Access Architectures The main memory itself is not organized as one monolithic block on modern multi-processor architectures. A single memory block

would constitute a bottleneck for several processors accessing it continuously. Instead, memory banks are physically distributed among processors, and a memory consistency protocol ensures that these distributed memories are seen as one consistent memory space to processes running on the platform. As a consequence, a processor has a shorter access time to the memory bank it is directly connected to, than to the memory banks connected to other processors in the system. Such architectures are thus named NUMA [57, 100], for Non-Uniform Memory Access. The relative placement of computations and data used by these computations on processors and memory banks respectively therefore impacts performance, and there usually is a benefit in favoring locality, that is, assigning computations on processors close to the memory banks they frequently access. However, other factors have to be taken into account as well. For instance, two concurrent computations having frequent synchronization steps should likely be assigned to cores/processors sharing some memory space (a memory bank or preferably a close cache); on the contrary, heavily concurrent memory-bound computations may benefit from targeting distinct memory banks to get a larger aggregated bandwidth. While the *hardware topology discovery* could be handled by computing runtime systems, the task is sufficiently heavy to justify a dedicated tool. This is the role of the `libhwloc` library [85]. In contrast, the task of deciding about which computing unit(s) and memory bank(s) to elect for some applicative code is a performance portability issue which a runtime system can contribute to address.

Discrete Memory Spaces Accelerator boards such as presented in Section 3.1.2 usually come equipped with their own embedded memory, distinct from the main memory space, meaning that data has to be transferred explicitly from the main memory to the board and back in order to use the accelerator. An approach very much alike this one is also adopted for some manycore processors such as the MPPA [61] from KALRAY or the SCC [96] from INTEL: in the MPPA, memory consistency is only enforced *within* each "compute clusters" comprising 16 processing engine cores, but not between the 16 compute clusters themselves; likewise, memory consistency is transparently enforced within each 2-core tile of the SCC, but not between the 24 tiles themselves. Thus, on these architectures, data transfers between cores belonging to distinct memory consistency domains must be performed explicitly by programs using them, as a design choice to simplify the hardware circuitry. In an opposite movement, instead, accelerators following the Heterogeneous System Architecture [130] (HSA) as well as some recent GPUs from NVIDIA support a unified memory system with the automatic management of data transfers between the main memory and the accelerator memory, relieving programmers from the task.

3.2 The Performance Portability Issue

The overview of high performance processor technologies in Section 3.1 shows the complexity, and hints at the numerous choices that applicative code developers face in porting and optimizing their code on a modern HPC platform. Many of these choices often have to be adapted or changed yet again when porting from a platform to another one. An important effort of the HPC community has thus been put over several decades to design tools to contribute to the portability of performance of HPC applicative codes. Enforcing portability for computation on HPC platforms involves multiple and often intertwined aspects: generating applicative code program for the targeted hardware instruction set, managing parallelism,

and driving the execution of the programs.

3.2.1 Writing and Compiling HPC Kernels

Multiple means are available for applicative code programmers to abstract away the writing or generation of code specialized for HPC computing hardware, such as general purpose processors with SIMD instruction sets, accelerators, GPUs, etc.

One option is to leverage compilers. Vectorizing compilers can translate programs written using "mainstream" languages such as C or Fortran into SIMD-enabled binary code in a fully transparent manner, provided the complexity of the source code structure remains tractable from the compiler. In more complex cases, however, a compromise is necessary, and the applicative code programmer has to get involved in the porting, for instance by using annotation-based language extensions of mainstream languages such as OpenMP or dedicated derivatives of mainstream languages such as OpenCL. In some situations, it can be necessary to resort to vendor-tied languages such as NVIDIA CUDA to access features not available through portable means, in which case the abstraction only holds among the hardware portfolio of the vendor. A variant of the compiler approach is the use of domain-specific embedded languages (DSEL) such as SYCL [155], embedded in C++.

Another approach, still related to compilers, is to use "intrinsics", which take the form of pseudo functions directly translated into specific processor instructions. This approach tends to be less portable than the first one because the set of intrinsics is usually tightly hardware-tied. It is usually reserved to hand-tuned SIMD ports of critical computing kernels or to write low-level code such as parallelism synchronization operations. To preserve some portability while still benefiting from fast intrinsics routines, a compromise is to utilise header-only libraries, also called "wrappers," such as MIPP [44].

If the applicative code uses well-known, widely used numerical kernels, it can also be possible to externalize the porting effort by leveraging HPC libraries of pre-tuned kernels instead of writing/porting them. This approach is typically used for common numerical computations, such as with linear algebra libraries (e.g. BLAS/LAPACK [7] implementations) or FFT (FFTW [79]). High performance implementations of these libraries and "hand-tuned" vendor-specific versions can be linked in place of the reference ones with little to no applicative code changes.

3.2.2 Expressing and Managing Parallelism

Beside the writing and compilation of applicative kernels themselves, a complementary aspect involved in exploiting HPC platforms is about taking advantage of the chip-level hardware parallelism. This encompasses the issue of identifying and expressing computing work requests that can possibly be processed concurrently without compromising the correctness of the program, and managing this parallelism to decide how to map it on available processing units, that is, to drive the execution of the program. Here also, multiple options are available to the applicative code programmer.

As a special case, if the strategy adopted for HPC kernels is to externalize them to numerical libraries, one possible way is simply to use parallel versions of these libraries if they exist. For instance the PLASMA linear algebra library is a parallel alternative to LAPACK targeting general purpose multicore processors; the CHAMELEON library [3, 1] is also a parallel alternative to LAPACK supporting multicore CPUs, accelerators and combinations

of them (and also distributed computing on multiple nodes). In this strategy, the work of expressing and managing parallelism is essentially hidden within the numerical library.

Also as a special case, when using an accelerator-oriented programming language such as OpenCL or NVIDIA CUDA, the parallelism among the computing units of the accelerator or GPU processor is managed implicitly. The kernel source code simply describes the routine that one computing unit has to perform, and this routine is then executed in parallel on the team of computing units participating to the computation within this accelerator or GPU processor.

On the lowest end of the abstraction spectrum, it is also possible to use low level parallel libraries such as `libpthread`. With this strategy, the applicative code programmer gets a full control of parallelism, though not on GPUs and not necessarily on accelerators either. However, the programmer also gets full responsibility for writing the parallelism management code and driving the execution of the program. The complexity of HPC hardware depicted above and the numerous trade-offs that have to be struck makes this challenging. Moreover, such an approach is undesirable from a software design point-of-view in that it results in execution control code interleaved with applicative code, mixing unrelated kinds of concerns.

Instead of compromising abstraction with `libpthread`, a better, rational, alternative in terms of software design, is to use a runtime system. This can be done either by expressing parallelism using the runtime system API directly, or by using a compiler to target the runtime system routines on behalf of the programmer. Again, for applicative codes with moderately complex structures, parallelizing compilers can compile the source code into a parallel-enabled binary code, otherwise, the programmer has to write the program using a parallel language; in both cases, however, the resulting binary is linked with a runtime system. Incidentally, the runtime-based approach is also used internally in many numerical libraries. The next sections study the design of runtime systems for computing, presenting issues and possible solutions.

3.3 A Thread-Based Runtime System for Computation: The MARCEL Library

The purpose of runtime systems for computing is therefore to handle the execution of HPC applicative code, so as to optimize it on the targeted hardware platform. Parallel programming is a difficult job for a human being, and as a result, the challenge is to design the programming model of a runtime system —that is, the abstraction— to make it tractable and convenient for the programmer, while preserving the efficiency and optimizing power at the execution model level. The awkwardness of reconciling these two objectives of abstraction and execution for computation runtime systems has led the community to propose and explore numerous design approaches over the now three decades of their development, resulting in a large diversity. We first investigate the thread-based programming model for computing runtime systems, based on the experience gained with the MARCEL [121] thread library, while the task-based programming model will be studied in Section 3.4.

3.3.1 The Thread Abstraction

The common trend of the early computing runtime systems, in the '90s, has been to propose programming variants around the notion of (software) *thread* [131] such as exposed by

the `libpthread` API. A thread is an independent flow of instructions within a process. It shares the process memory space with the other threads, but maintains its own state, and usually its own stack —but not always, since the lightweight threads in the Filaments [112] framework are stack-less—. With these programming models, a program expresses parallelism by spawning threads, that is by creating new instruction flows, to perform computation work.

In the corresponding execution model variants, threads are assigned to computing units under the control of some scheduling algorithm. The scheduler may either be cooperative or preemptive depending on the runtime system implementation. According to the cooperative approach, a thread assigned to a computing unit remains assigned to this unit until it terminates or temporarily blocks on some condition, upon which the scheduler may assign another thread to this unit. According to the preemptive approach instead, the scheduler may forcibly interrupt and temporarily pause a running thread after some quantum of time has elapsed, to schedule another thread on the same computing unit. In both cases, the role of the scheduler is to "multiplex" the thread work requests on the computing units. If the scheduler belongs to the operating system itself, the thread model is deemed a *system-level* thread model; otherwise the model is deemed an *user-level* thread model. The overall scheme can be recursive: a thread may itself host an inner thread scheduler, in which case the execution time of the outer "host" thread is shared among one or more inner threads by the inner thread scheduler.

3.3.2 The MARCEL Thread Library

Such a two-level multithread architecture was adopted for the MARCEL multithreading library [121]. MARCEL was initially designed by Raymond Namyst during his PhD Thesis [120], then further developed at LIP Laboratory in Lyon, and eventually by the team Runtime of LABRI laboratory and INRIA Bordeaux – Sud-Ouest. I was maintainer of the MARCEL library from 2008 to 2012 and supervised the implementation of the POSIX PTHREAD compliance API on top of it as part of the INRIA VISIMAR development action.

The MARCEL library is organized as a two-level thread architecture where programmer-facing user-level threads are scheduled on top of a pool of system threads. The system threads are launched at startup time and bound to CPU cores. Thus, assigning user threads to a given system thread amounts to deciding on which core the user thread is going to run. Each user-level thread is preemptible by MARCEL's thread scheduler. Preemption may occur on a blocking condition or when the quantum of execution time allocated to the thread expires.

After the work and PhD thesis of Samuel Thibault [165, 164] on scheduling on top of NUMA multicore platforms, the MARCEL scheduling engine has been extended to offer multiple queues (named *runqueues*) for queuing threads, following the machine topology: one queue encompassing all cores, one individual queue dedicated to each core, and intermediate queues encompassing relevant sub-sets of cores, such as cores sharing some level of cache or some NUMA memory node. A thread assigned to a runqueue may be executed on any core owning the runqueue, possibly a different core after a preemption context switch.

This hierarchical set of runqueues enables the programmer to control the level of affinity to be enforced for each thread: assigning a thread to the runqueue of a narrow set of cores, and even more to an individual core runqueue, favors affinity and locality. However, if some core outside the narrow runqueue becomes idle, it cannot grab threads from this runqueue to execute them, which may possibly create imbalance. On the contrary, a user thread in a broad runqueue favors load balancing, at the cost of affinity and locality: a user thread from such a runqueue may get scheduled on many different cores throughout its lifespan, causing

expensive cache misses and possibly NUMA memory access penalties as well.

3.4 A Task-Based Runtime System for Computation: STARPU

We now investigate the task-based programming model for computing runtime systems, based on the work with the StarPU runtime system. The task parallel programming model was initially popularized in particular by the CILK [80] programming environment and associated runtime system in the nineties. It became the focus of a renewed interest around the mid-2000s decade, in the wake of emerging multicore processors, accelerators, general purpose GPUs, and the "sudden" re-orientation of the hardware computing vendors' strategy towards increasingly massive chip-level parallelism, still followed today.

3.4.1 The Task Abstraction

The concept of *task*, as the common denominator of the numerous task-parallel programming model variants, designates "some work" to be executed by some pre-existing, "persistent" worker thread; "*some work*" meaning a user supplied function or region of code depending on the model specifics; "*persistent* worker thread" meaning that the lifespan of such a worker thread will usually largely exceed the lifespan of a single task, so that it can be reused to execute other tasks.

In practice, a team of worker threads will typically be launched at startup time, or when entering a parallel processing portion of the applicative code —depending on variants of the model— and this team will execute a set of tasks supplied by this applicative code until all these tasks are completed. The key idea of task-based programming models is that a task only specifies a piece of work to be realized, but not the resources to realize it. These resources are instead supplied by the team of worker threads. Thus, task-based programming models are designed to rationalize the management of the work realized by the threads, while thread-based models leave this management under the responsibility of the user code.

Many variants of the task-based parallel programming model have been proposed over time. The varying characteristics of these models include:

— Whether a task is allowed to block —for instance waiting for a lock or for an event— or must run to completion in one go;
— Whether a task is allowed to create new tasks —and to wait for their completion, in which case this implies the previous characteristic as well— or not;
— Whether the tasks may be inter-dependent or are all independent of each other;
— If they are inter-dependent, how the dependences are expressed, through task–task dependences, or through task–data dependences, or through opaque objects, or even through a combination of these three kinds;
— How/when the tasks are discovered, either through a compilation step [36], or on the fly at run-time.

In many ways, the concept of task can therefore be seen as an extension of the notion of *function* to the context of parallel programming, and these varying characteristics of task-based parallel programming models correspond to the exploration, by the community, of what can be possible and/or achieved when some aspects are supported or forbidden. For instance, models that let tasks create children tasks and wait for their completion basically allow the property of *recursivity* on tasks, as is the case with CILK: this aspect has been

successfully used by CILK on the *divide-and-conquer* class of problems [31], and to develop *cache-oblivious* algorithms [160].

In particular, enabling the notion of dependence on a task model gives the programmer the opportunity to indicate ordering constraints that should be met by the runtime system in deciding in which order to execute the tasks. Such dependence constraints can be expressed either explicitly, as task–task dependences, or implicitly through data dependences or proxy object dependences. Moreover, if the task model forbids tasks to block, it means that tasks in such model cannot have any hidden dependence to the runtime system view. Together, a model of task letting the programmer express task dependences *and* forbidding tasks to block creates a task concept very close to the notion of *pure function* in the domain of functional programming.

In functional programming, a pure function is a function that has no side effect: all its interaction with the "outside" world is carried by its input parameters and its output result(s), which enables the associated language interpretor/runtime to evaluate the function lazily —and possibly not at all— without inconsistent results. A non-blocking, interdependent task model is similar: all the task interactions with the outside world are carried by the expressed dependences, which enables the runtime system to generate tasks execution orderings — that is, tasks' *schedules*— without any risk of deadlock [1]. The STARPU runtime system [13] presented next builds on these properties of the non-blocking dependent tasks variant of the task parallel programming model to offer a powerful and versatile task scheduling environment for heterogeneous HPC platforms.

3.4.2 The StarPU Task-Based Runtime System

In 2008, I submitted a proposal to the French national agency for research (ANR) of a 3-year (2009–2012) project named PROHMPT (Programming Heterogeneous Multi-Processing Technologies), gathering French teams from INRIA, CEA, the University of Versailles, as well as companies BULL (now part of company ATOS) and CAPS ENTREPRISE, on the hot topic at the time of designing new programming models, environments and tools for the emerging heterogeneous computing platforms equipped with GPUs or accelerators. The project was funded by ANR [2], and one of its outcomes has been the StarPU task-based runtime system [13], initially primarily developed by Cédric Augonnet as the centerpiece of his PhD Thesis [12] and by Samuel Thibault, one of his advisors. I have personally contributed several parts of the StarPU runtime since then, such as the global synchronization scheme of the StarPU core engine currently in use, the adaptation layer implementing the semantics of the OpenMP parallel language on top of StarPU, the StarPU's native Fortran interface, and the resource interoperability layer with external runtime systems, among others.

The STARPU runtime system has been designed from the start to provide a task oriented parallel programming model targeting multicore nodes of HPC platforms, optionally equipped with accelerator boards or general purpose GPU boards. As discussed in Section 3.1.2 introducing accelerators, one of the major challenges posed by this kind of hardware is to decide whether a piece of work (e.g. a task) for which a CPU and an accelerator im-

1. One can argue that some deadlock may result from a runtime generating a schedule resulting in an out-of-memory condition, while another schedule consuming less memory would have been possible to prevent that condition. However, waiting for some memory to become available is actually a dependence, which should be expressed explicitly if such an out-of-memory condition is at risk to happen [147, 25].

2. Grant ANR-08-COSI-013.

plementation are available should better be executed on the accelerator or on the CPU. The decision naturally depends on the respective execution time of the task on a CPU resource or on an accelerator resource, but not only. It also depends on the cost of data transfers (data input and/or result output) that might be incurred for using one resource vs the other, and possibly on the other tasks being scheduled as well. Thus the main idea of the STARPU runtime system is to associate:

— A performance modeling framework to estimate the expected execution time of a task on each of the available heterogeneous computing resources for which an implementation of task is available, and to estimate data transfer costs on system buses;
— A distributed shared-memory manager in charge of handling data replication and consistency between the main memory space and the memory space(s) of the accelerator board(s);
— A task-based programming model leveraging the properties of the dependent, non-blocking tasks variant to enable generating task schedules in a safe manner, thanks to the fully expressed dependence constraints, and in an efficient manner, thanks to the performance models and the memory manager.

We detail these characteristics further below, to examine the design choices, their properties and limits.

Programming Model The reference programming model of StarPU is called the *Sequential Task Flow* (STF). It is derived from the "non-blocking tasks with dependences" family of models, with the additional particularity that tasks *should* be submitted to STARPU as part of a unique sequential task flow, as the name implies, that is, by a single application thread. This is however not a hard requirement (hence the use of 'should' and not 'must'). Yet, submitting tasks from multiple concurrent flows may result in a non-deterministic behaviour, if done incorrectly. Thus, we will assume in the following that tasks are submitted as part of a sequential flow, except where explicitly mentioned otherwise. Also, while STARPU supports task–task dependences and task–"proxy object" dependences (named *tag* dependences in the STARPU terminology), its reference STF programming model only employs task–data dependences, and we will only consider these here for concision.

The reference STF programming model of STARPU works as such: tasks are submitted by one thread of the applicative code one at a time. A task submission specifies which computation kernel is to be realized by the task (e.g., what piece of work to do), which pieces of data are read by the task, as well as which pieces of data are written to. The computation kernel of a task is indicated by passing a *codelet* object to StarPU's task submission routine; this codelet object lists all the available implementations of the computing kernel from which STARPU will attempt to select the most appropriate one during the scheduling process. Each piece of data referenced by the task is specified by passing a corresponding *data handle* in the STARPU terminology, with an access mode specification (e.g. read, write, read/write). From this flow of tasks with data dependences, STARPU builds a directed, acyclic graph (DAG), where the tasks are the vertices, and the edges are dependences, through dependence analysis [28]. When a new task $\mathrm{T}_{\mathrm{new}}$ is submitted, a new vertex is added to the graph. When a conflicting "read-after-write" (RAW) dependence on a piece of data is detected with the most recent task in the task flow, an edge is added in the graph between the new task and the task from which it depends. The "write-after-write" and "write-after-read" dependences are also considered but can be eliminated through the *renaming* technique. The DAG of tasks

basically summarizes all the constraints that the StarPU scheduling engine must take into account to produce valid schedules.

The STF model is directly inspired by the instruction reordering technique used in superscalar processors (see Section 3.1.1): the data dependence-based method to determine valid processor instruction reordering operations is used here to determine valid schedules from the task graph. This technique is also employed by other runtime systems, and in particular by the StarSs/OmpSs family or runtimes developed at the Barcelona Supercomputing Center (BSC), from which the family draws its *SuperScalar* (Ss) suffix name.

Memory Model On a heterogeneous computing node equipped with one or more discrete accelerator boards, the main memory and the memory areas embedded in the accelerator boards are distinct. Thus a piece of data must be present in an appropriate memory area prior for a task to access it. Some hardware supports mapping the memory of an accelerator board into the process main CPU memory space; however, even in that case, a data transfer to the right memory area prior to executing a task will often be more efficient than letting the task start immediately and access the piece of data remotely while computing. Moreover, unnecessary data transfers may be eliminated by keeping data replicates in several memory areas as long as they remain consistent.

The management of data transfers between memory areas is handled by StarPU's distributed shared memory (DSM) engine. Its major role is to make sure that all required pieces of data are available or made available at an appropriate location before making a task "ready" for execution, using prefetching techniques when possible to avoid the task execution being delayed. It implements replicates management and redundant transfers elimination through the MSI [127] (`Modified/Shared/Invalid`) cache consistency protocol.

The topology of the computing platform is detected using `libhwloc` [85], to detect any NUMA memory layout, and to discover the connectivity between the main processor(s) and any accelerator board. The topology map is then used by the DSM engine at startup time to collect performance metrics regarding the cost of transferring data from one memory area to another one of the platform, generically referred to as the *bus performance*, and supply the information to the performance modeling framework.

Execution Model StarPU's execution model is asynchronous with respect to the application thread submitting the sequential task flow. Execution of tasks is performed by a team of worker threads launched at start-up time. By default, one worker thread is launched for each CPU core, and among those worker threads, one worker per accelerator device is dedicated for monitoring that device instead of executing tasks. Thus, on a computing node having 8 CPU cores and 1 accelerator, StarPU will use 7 worker threads to execute tasks on the CPU, and 1 worker thread (and thus, 1 CPU core) to drive the accelerator, the rationale being that given the raw performance ratio between an accelerator and a single CPU core, it is usually beneficiary to dedicate that core to feed the accelerator with new tasks as quickly as possible. The application thread submitting tasks to StarPU is not considered a part of StarPU's worker threads team, it is therefore not used for running tasks.

A task goes through four states during its lifespan: `submitted`, `ready`, `scheduled`, and `executing`. It transitions from `submitted` to `ready` only after all its incoming dependences have been resolved. The `ready` state marks the point in time where a task becomes considered by the StarPU scheduling engine. While the scheduling process could theoretically start

right after the task is `submitted`, the cost and software design complexity were deemed too high for an uncertain benefit. The choice of defining distinct `scheduled` and `executing` states enables tasks to be anticipatively assigned to per-worker thread queues. Once a task is assigned to worker queue, waiting for the worker to execute it, data prefetching operations can be launched so that, when the worker eventually dequeues the task to execute it, it can run the task immediately if the prefetch operations have completed in time.

The scheduling process is handled by programmable policies. STARPU ships with a series of pre-defined scheduling policies. New policies can be supplied by the applicative code to extend this set with customized scheduling algorithms, thanks to a dedicated programming interface. Conceptually, `ready` tasks are *pushed* by the applicative code thread submitting tasks, and *popped* by the worker threads; everything in-between the *push* and the *pop*, including the choice of data structures to queue tasks, is left under the responsibility of the selected scheduling policy. Such a design is particularly flexible in terms of the kinds of scheduling policies it enables: for instance, it makes it possible to implement *proactive* policies, where the scheduling work mainly occurs on the *push* side to anticipatively assign tasks to workers using some planning algorithm from the rich corpus of theory on task scheduling; to implement *reactive* policies, where the scheduling queue mainly occurs on the *pop* side using work stealing techniques; or even to combine both, where the *push* side performs an initial assignment of tasks to workers, which can be further adjusted on the *pop* side to correct residual load imbalances on the fly.

Adaptiveness, Modularity, Tuneability, Extensibility Bringing portability of performance to applicative code developers requires from runtime systems to adapt to a variety of existing hardware platforms, as well as to platforms likely to emerge in a foreseeable future. Beside using an abstraction layer and a set of device drivers in a typical runtime system fashion, STARPU incorporates several design features aimed at ensuring the portability of performance property, and to make it possible to prolong that property as evolving needs arise, while avoiding expensive in-depth redesigns.

A performance modeling framework is integrated within STARPU to collect quantitative informations about the duration of tasks on the various computing units for which an implementation is available. The collected data is used to build performance models of tasks, that is, to infer the execution time of "similar" tasks subsequently scheduled on the same hardware computing resources. The scheduling policies can, optionally, exploit such performance models to decide how to map tasks on the available computing units. This, however, assumes that the execution time of a modeled task is stable with respect to some parameter (usually the input size, but not necessarily). Tasks whose execution time depends on the actual contents of input data, such as iterative kernels targeting some threshold or lookup routines, cannot be modeled through this framework in the general case. Scheduling policies can combine task performance models with the bus calibration information from the memory manager (see the "Memory Model" paragraph above), to take into account the cost of data transfers from one memory location to another one in the scheduling decision.

Many aspects are made tunable or customizable: The bias weight between the task performance model and the bus calibration model can altered in favor of load balancing (by decreasing the weight of data transfers relative to task execution times) or locality enforcement (by making data transfers more expensive); custom task performance functions, custom data layout management functions, custom scheduling policy functions can be supplied by the

applicative code to handle new use cases. Performance models for tasks, stored across sessions to save time, are automatically refreshed when a significant divergence is observed between a task execution time inferred by the model and its actual measured execution time, so that any evolution in the platform, either on the hardware or the software side, can swiftly be taken into account.

Separation of Concerns Portability of performance is helped further through the notion of *separation of concerns* promoted by runtime systems. In the case of StarPU, at least five separate concerns can be identified and handled by possibly different specialists:

— The applicative code core part;
— The task kernels;
— The scheduling policies;
— The device drivers;
— The runtime system itself.

The core applicative code and the task kernels, for instance, need not necessarily be written by the same person. Instead, the core STF algorithm can be written by a specialist of the targeted application field (e.g. perhaps a physicist or applied mathematics specialist for a physical phenomenon simulation application, for example), while the task kernels may perhaps be written by computing optimization specialists, or even by vendors in mathematical libraries optimized for their computing hardware.

The scheduling policy can be selected independently of the applicative code and tasks, and new policies can be implemented also, for instance by specialists of the task scheduling algorithm community. This property also enables experimenting with multiple policies without any modification on the applicative code and on the tasks, and no modification required at the level of the runtime system either, making it easy to compare and select the most appropriate one.

The device drivers handle most of the work of initializing accelerator devices and interacting with the specific programming means of such devices (e.g. Nvidia Cuda, OpenCL, etc.) to execute tasks on them, manage on-device memory allocation and handle data transfers.

Indeed, the runtime system by itself remains independent of these other concerns, as long as the programming interface stays backward compatible. This way, it has a significant latitude to evolve for accommodating new hardware capabilities or address new needs without breaking scheduling policies and drivers, and most importantly without breaking existing applicative codes and task implementations. This property was successfully leveraged to integrate functionalities demanding heavy internal developments without impacting user codes, as can be seen with some of the examples below:

— The *scheduling contexts* developed by Andra Hugo as part of her PhD. Thesis [97, 55], enabling multiple applicative codes to be scheduled concurrently within the same StarPU session while finely and dynamically controlling the computing resources assigned to each code.
— The OpenMP compatibility layer of StarPU, discussed further in Chapter 5, enabling capabilities such as blocking tasks and recursive tasks as required by the OpenMP programming model, while preserving the existing programming and execution models for native StarPU codes.
— The implementation of interoperability capabilities of the StarPU runtime with the OmpSs runtime in terms of computing resource sharing (see Chapter 6). For

instance, this enabled an OMPSS application code to call whole parallel routines from the CHAMELEON linear algebra library [1] (that is, routines implemented using STARPU tasks) from within OMPSS tasks, without requiring any modification inside the CHAMELEON library.

3.5 Discussion

Through the works on the MARCEL library and subsequently on the STARPU library, we have explored a substantial part of the domain of computing runtime systems, from thread abstractions to task abstractions, from recursive parallelism, nested parallelism, to flat parallelism, and to dependent task parallelism.

3.5.1 Some Level of Maturity

The general impression on that domain is that some level of maturity seems to have been reached. Through the use of well defined paradigms, such as the *Parameterized Task Graph* model (PTG) used by the PARSEC environment, based on compact DAG representation [56], or the *Sequential task Flow* model (STF) and variants used by STARPU and others (OMPSS [40], SUPERGLUE [166], also PARSEC, more recently [95], ...), based on dynamic task discovery and dependence analysis [28], modern runtime systems are "reconciling" theory and practice for new classes of parallel applications, beyond the success story of the CILK environment [80] on divide-and-conquer problems. In particular, the properties of the STF model combined with STARPU's programmable scheduling engine enable implementing, integrating and experimenting with a broader range of scheduling algorithms than only work stealing algorithms, and to exploit relationships between potentially parallel pieces of computation. Moreover, the large number of task-based runtime systems proposed during the last decade seems to indicate some kind of consensus among the HPC computing runtime system community in favor of the task abstraction for user-facing parallel APIs —except for kernels where parallel loops are sufficient— even if this large number of task runtimes also shows a lack of consensus about the model details, features and expressiveness.

The thread abstraction also appears to have a reached some maturity. On the general purpose, system thread API side, the use of the POSIX PTHREAD API is ubiquitous, while a number of initiatives have emerged on the light-weight, user-level thread API side [145, 172, 50], with even some unification initiative among them [49]. The positioning of the thread abstraction in HPC as evolved over time, from directly facing the applicative code programmer to being rather used as a building block for higher level programming models and environments. It is now mainly used through OPENMP parallel regions and loops, through task-based frameworks, and also for supporting concurrent services such as network communications management and event monitoring. In the end, the relative positioning of user-level thread APIs and task-based programming environments appears more a matter of point of views nowadays than a matter of technical characteristics: on the one side, task-based environments may allocate stacks on a per task basis for supporting potentially blocking tasks, as is the case with STARPU's OPENMP runtime support (see Section 5.3), while on the other side, user-level thread packages may provide stack-less thread objects beside regular threads, as the *tasklet* objects in [145]. Instead, the distinction is more about whether these technical details are exposed to the applicative code programmer in the case of user-level

thread packages, or abstracted away in the case of task-based programming environments.

3.5.2 Granularity and Overhead

Yet, even though the task-oriented abstraction is reaching a level of maturity, some issues remain. The question of selecting the appropriate *granularity*, that is, the right computational weight, comes among the prominent issues of task-oriented parallel programming, and is discussed in more detail in Section 8. If the applicative code is split into too fine grained tasks, the ratio of tasks management costs (data structure allocation, queuing, scheduling, etc.) over the computation load becomes prohibitive, while, on the opposite, selecting too large grained tasks may harm load balancing due to the lack of flexibility in scheduling. Moreover, this "right" computational weight highly depends on many parameters, both on the hardware side and on the software side, which may require some calibration process. The applicative code itself may or may not accommodate well to this need to supply the right computational weight. For instance, kernels with very lightweight computation elements or showing little load imbalance in the first place may be better served with parallel loops instead of task parallelism, since the management cost of parallel loops is much lower than the management cost of parallel tasks.

3.5.3 Scalability

As the number of cores and hardware threads increases with each new generation of machines, the scheduling engine of a computation runtime system may become a bottleneck, struggling to feed each computing unit with sufficient work to perform. Several techniques have been explored to keep the scheduling cost at a tractable level on these many-core and dense multi-core platforms, such as nesting schedulers from a coarse level to a fine level, so that the first level scheduler performs a rough work assignment to groups of computing units while nested schedulers refine the assignment to increasingly smaller groups of units [40] and eventually to individual units, possibly using distinct runtime systems at distinct levels as discussed in Section 6.2. The work of Terry COJEAN [54, 55] on node clustering and parallel tasks also aimed, among several objectives, at improving scalability on many-core platforms such as the INTEL KNL. It proposed to schedule so-called *parallel tasks*, that is scheduling tasks running on more than a single core at a time. For that, it clustered computing units into logical groups, in a systematic, topology-aware manner (e.g. a group of cores sharing some cache level or some NUMA bank) to reduce the scheduling cost, at the expense of potentially increasing the overall load imbalance. Other works such as the BUBBLESCHED mechanism developed by Samuel THIBAULT [164, 165], and refined more recently by Pierre-André WACRENIER on bubble scheduling with tasks, aim at proposing programming abstractions to express multiple views of parallel entities that can be selected lazily at run-time. Such bubbles may offer a mono-entity view for reduced scheduling cost, a multi-entity view (a nested DAG) when increased parallelism is needed to feed idle units, and possibly additional alternative views as well. Such views can be seen as an extension of the concept of multiple kernel implementations supported by STARPU codelets, and could eventually (though this will be challenging!) be performance-modeled to help selecting views automatically.

3.5.4 Adoption

In line with the granularity and scalability issue, the work of refactoring existing applications into task-based codes is also a major hurdle to the adoption of task parallelism paradigms by applicative code programmers. In many cases, this refactoring step is not straightforward. It requires re-thinking both the algorithmic aspects and the data structure aspects of the code. It requires identifying and declaring any dependence explicitly. And it cannot always be done in an incremental manner. In order to ease this process, the use of higher level approaches can help, for instance using specialized languages and frameworks, as for instance the INKS environment discussed in Chapter 7. Parallel languages such as OPENMP, explored in Chapter 5, can also simplify the refactoring process to some extent, making it easier to switch from a paradigm to another; though even so, converting an OPENMP application to task parallelism may involve much more than just renaming a few pragmas. Thus, there is likely a need for some tools and guidelines to facilitate the "taskification" process. The work presented in Chapter 9 on the use of the MAQAO tool to help exploiting SIMD instruction sets could be extended to guide an application programmer in delineating tasks in existing codes.

3.5.5 Debugging

Likewise, there is a critical need for debugging tools and especially on performance debugging tools to let end-users investigate when things go wrong or the performance obtained appears suboptimal, while a computing runtime system is involved. Since a key role of runtime systems is to defer many decisions at execution time, the process of investigating issues requires being able to know which decisions are taken, and for which reasons. Some tools and mechanisms already exist, typically trace collection and Gantt chart display tools (for instance the FXT/VITE tools used with STARPU, or the PARAVER tool used with OMPSS), performance analyzers such as INTEL VTUNE, or the task-oriented debugger TEMANEJO [102]. The HELGRIND tool from the VALGRIND toolchain can also detect consistency and locking issues in multithreaded programs. Yet, interpreting the output of these tools and determining the root cause of an issue can be challenging even for experts in parallel programming. There are several reasons for that. Due to the separation of concerns enforced by runtime systems, a given bug or performance issue may be due to one of the software elements involved (the applicative code, the runtime system engine, the scheduling policy, some device driver, etc.) or a combination of these, or even the interplay of these software elements with the hardware topology (e.g. some cache conflict or memory contention). Moreover, the traces generated by the tools can quickly get huge due to the time scales involved and the level of concurrency. Thus, there is certainly a need to improve the feedback mechanisms available to programmers, to simplify and automatize the analysis process, perhaps by leveraging statistical tools to detect anomalies in execution flows.

Also, some issues, especially in terms of consistency, can be avoided by a careful preventive analysis of the source code. Software tools such as the PARCOACH framework [141, 140] are designed to automatize this kind of analysis, to detect that, for instance, not all the threads of a thread team can reach a barrier if the execution of these threads may diverge on some branch instruction. Such a class of tools could be extended, for instance to detect that task dependences are accurately declared, and that no hidden dependence remains.

3.6 Conclusion on Performance Portability of Computations

In this chapter, we investigated performance portability from the point of view of two runtime systems for computing, the MARCEL user-level thread library, which I first used and then maintained as part of my research work, and STARPU, a task-based runtime developed by Cédric Augonnet during his PhD thesis, for which I am now one of the maintainers. We saw that computing runtime systems divide into two main classes depending on whether they let the applicative code express parallelisms in terms of workers —for thread-based runtimes— or in terms of pieces of work to be done —for task-based runtimes— and we explored both of them from the study of the design of the MARCEL and the STARPU runtimes. We saw that task-based runtimes have gained a lot of traction, especially after the arising of multicore processors, thanks to the properties of task objects making them more easily subjected to scheduling algorithms; and among them, the strength of dependent task-based models, due to the additional semantics available to drive the execution. However, we also observed that there is also a more recent regain of interest for thread-based runtimes especially as a low-level support for implementing other models. We discussed open issues and perspectives, in terms of granularity and scalability, but also in terms of user friendliness to convince more developers to rely on task parallelism, and give them tools and methodologies to analyse performances.

Next chapter will now explore the combination of computing and communication supports within an unifying and consistent task-based programming model.

Chapter 4

Performance Portability of Distributed Computations

Having runtime systems for communication and runtime systems for computation means that a combination of two runtime systems should be needed to fully utilize a HPC cluster. This explains the rather widespread use of the "MPI+X" class of models in HPC applications, meaning that such applications use MPI for networking and distributed computing management *between* nodes, while a computing runtime system "X" is used for managing parallel computing at the node level, with "X" often being an OpenMP runtime.

Some parallel+distributed applications also use the MPI-only model, where one process is launched for each CPU core and MPI routines are used both for inter-node communications and for intra-node communications through shared memory operations. This model may appear simpler due to the use of single API. However, it requires more work to handle node-local interactions, it wastes some resource due to the larger number of processes. Also, it offers less control over resource management and work scheduling among the multiple processes launched on each core within a node.

Yet, having a single user-facing programming interface instead of two is desirable from the point-of-view of making it easier to write parallel+distributed applications. The idea of *integrating* a computing runtime and a communication runtime within a single, consistent model is therefore interesting in that it enables this simplification through an unified API, while offering to make both runtimes actually cooperate with each other, instead of being merely juxtaposed in isolation. Moreover, among computing runtime systems, dependent-tasks oriented models such as the one offered by StarPU bring some unique additional opportunities for building distributed computing models, thanks to the data dependence constraints expressed on tasks. This is the approach we study in this chapter.

4.1 Integrated Parallel and Distributed Programming

Most, if not all, HPC applications need to use *parallel* computing and *distributed* computing together to obtain a sufficient level of aggregated performance, the "HPC" tag naturally being the indication of such outstanding computing resource requirements. The "MPI+X" class of models, often used by HPC applications to exploit parallel and distributed computing together on supercomputing clusters, consists in associating the use of MPI for programming the distributed computing management part, and some parallel programming model and

associated runtime system for the node-local parallel computing part. A distinctive aspect is that the association between the MPI implementation runtime and the "X" parallel model runtime is informal: both models are programmed through distinct interfaces within the applicative code; moreover, neither the MPI runtime nor the "X" runtime are made aware of each other. Making both work smoothly together is challenging and involves taking technical details into account that should not be of concern for a scientific application programmer, such as making sure that no deadlock occurs, that communications steadily progress and are properly overlapped with computation whenever possible, that communication threads within the MPI implementation runtime do not interfere with worker threads in the computing runtime, just to mention a few of the most prominent issues. As a result, application developers typically bring the complexing back to a tractable level by strictly alternating computing and communication phases using synchronization barriers, such that only at most one of the two runtimes is in use at any time. While this scheme may fit highly regular HPC applications well, it imposes an awkward artificial algorithmic layout in the general case, and causes performance degradation on less regular applications due to the resulting idle time at synchronization barriers.

MPI-only HPC applications exclusively rely on MPI routines to manage inter-nodes and intra-node interactions. This scheme remains popular also, especially for applications that pre-existed the advent of multi-core CPUs in the mid-2000s, because it is straightforward to use: one merely needs to launch the MPI session with one process per CPU core instead of one process per node or per NUMA node with MPI+X. This model is not so straightforward to use efficiently, however. Since processes do not share data by default, some extra programming effort with shared memory management may be needed to avoid redundant memory allocations for shared data structures residing in distinct processes of the same node. Moreover, applications using this model essentially run without any computing runtime system, thus they exclusively rely on the operating system and process binding commands for computing resource management, work mapping and load balancing. Besides, ensuring that computations and communications progress concurrently entirely ends up under the responsibility of the application developer.

Integrating a computing and a communication runtime systems in a coordinated manner under a unifying programming interface therefore presents a number of benefits over the "do-it-yourself" MPI+X and MPI-only approaches. On the programming model side, an unified API enables to hide the complexity of managing parallel and distributed computing together [110, 149, 175, 40, 162], and even to abstract inter-node exchanges entirely [30, 176, 101]. On the execution model side, obtaining a comprehensive and consistant view of the application overall pattern, instead of split views of computing patterns and communication patterns, opens the opportunity of a more rational and accurate resource management. In this respect, the possibility of using a DAG of tasks to represent the application pattern as a whole, depicting work and data relationships together, makes the *dependent tasks* parallel programming model a prominent option for a unified parallel+distributed runtime system. This is the option that we investigated with the StarPU runtime system [3] as part of the PhD Thesis of Marc Sergent [146], which I co-advised. We explore this option below.

4.2 Extending the STARPU Task-Based Runtime Systems for Parallel and Distributed Computing

The STARPU runtime system, presented in Section 3.4 in its original single-node form, already supports "some kind" of distributed computing when used on a node equipped with an accelerator boards. In a master-slave manner, the STARPU instance on the host —the *master*— drives computations on the accelerators —used as *slaves*— either through dedicated runtime systems such as CUDART for NVIDIA CUDA GPU boards, or through a slave instance of STARPU on discrete manycore accelerators such as INTEL's KNC boards. However, for aiming at running on the kind of distributed clusters found in nowadays supercomputing centers, a master-slave scheme would cause a bottleneck at the master. The distributed variant of the OMPSS runtime system [162] addresses this issue using a nested multi-level master-slave scheme, where a top-level master shares bulk pieces of work among some slaves processes; they in their turn partition their own piece of work into finer-grained pieces assigned to a nested level of slaves, for which they act themselves as masters. However this nested scheme may be artificial to the applicative code fundamental algorithm, and it breaks the sequential flow of tasks into multiple segments. We thus selected a distinct approach instead, which consists in combining the unmodified *sequential task flow* programming model with a *fully distributed* execution mode, along the line of the QUARK runtime system distributed extension [174], though with additional procurements to enhance scalability.

4.2.1 Distributed Sequential Task Flow Programming Model

The distributed version of the programming model preserves the principle of dynamic discovery of the sequential task flow: all participating processes submit tasks, following the same sequential algorithm, to their local STARPU instance. In addition to the single node version, however, the applicative code also gives an initial distribution of data among the distributed nodes, specifying the ownership of data pieces among the nodes. Thus, conceptually, the same unique graph of tasks, which expresses the sequential consistency requirements that should be enforced, is submitted by each participating process. Moreover, the programming model does not involve designating any master process; instead, all processes have an equal role with respect to the session. Thus, adapting an applicative code from the single-node programming model to the distributed programming model of STARPU is a breeze.

4.2.2 Fully Distributed Execution Model

The key idea is that the *Fully Distributed* execution model entirely and exclusively relies on *local* decisions only; there is no global scheduler involved. While all processes share the same task graph, each participating process executes its own specific subset of the graph. A process locally determines, by itself, whether to execute a given task or not. It does so according to the ownership of the piece of data the task writes to —by default— as given by the data distribution: if the piece of data the task writes to is owned locally, the process elects to execute the task. Additional rules are available to resolve ties when multiple pieces of data are written to by the task, and alternate policies can be selected also.

Required data transfers are automatically inferred by STARPU. When the local STARPU instance of a process encounters a task that it will not itself execute, but for which a piece of

data on which it depends for execution is determined to be locally owned at this point in the DAG, that instance inserts a send communication request along the graph dependence edge. Likewise, the process that will execute this task —which thus needs that piece of data as an incoming dependence— will generate a recv communication request along that same DAG edge. Thus, dependence edges linking two tasks executed by distinct processes are served by network requests, while edges between two tasks executed on the same process are served as in the single node version of StarPU.

The execution model therefore basically relies on the fact that each process runs the same state machine and goes through the same sequence of states as given by the sequential flow of tasks: The fact that all processes essentially get the same "roadmap" is the property that enables each process to locally take decisions that are consistent with the decisions of the other processes.

4.2.3 Two Distinct Frames of Reference

The sequential task flow model involves two very distinct "frames of reference": the *submit* frame of reference, and the *execution* frame of reference. While this is also true for the single process version, the consequences are more visible in the distributed version.

The *submit* frame of reference designates a virtual timeline along which all "operations" are ordered along the logical order in which they are submitted, which serves as the reference for sequential consistency; the word *operations* here encompasses tasks, communication requests for inter-process dependence enforcement, and more generally any state change in a conceptual *global* state machine. This means that, for instance, a send communication request should not actually be started before the task that produces the piece of data to send has completed —thus, a communication request follows the same ordering rules as a regular task—. Moreover, an operation such as an explicit data migration request by the applicative code, whose action is to change the ownership of a piece of data, should also be ordered with respect to the *submit* frame of reference, so that all processes get a consistent view of the change.

The *execution* frame of reference designates the real "space-time", in which operations are actually realized along any weakly ordered manner compatible with the constraints expressed at submission time in the *submit* frame of reference. It is instead asynchronous with respect to the main applicative code, and hence mostly decoupled from the *submit* frame of reference. As a consequence, the main applicative code should not interact directly with the *execution* frame of reference, except through the specifically supplied StarPU methods such as for waiting for the completion of submitted works. Otherwise, such an interaction would introduce a hidden dependence, that is a dependence not carried by a DAG edge, which would break the principle of exclusively using "pure function-like" routines in the non-blocking dependent task model, as discussed in Section 3.4.1.

4.2.4 Involving the Applicative Code in the Optimization Process

Global Task Graph Pruning While the exclusive use of local decisions brings a major advantage for the *fully distributed* execution model in terms of scalability, due to the elimination of the bottleneck found in master-slave models, it comes with a significant counterpart in that the whole task graph must be discovered by each process. This means that increasing the number of tasks, for instance to work on a larger application problem and on a larger

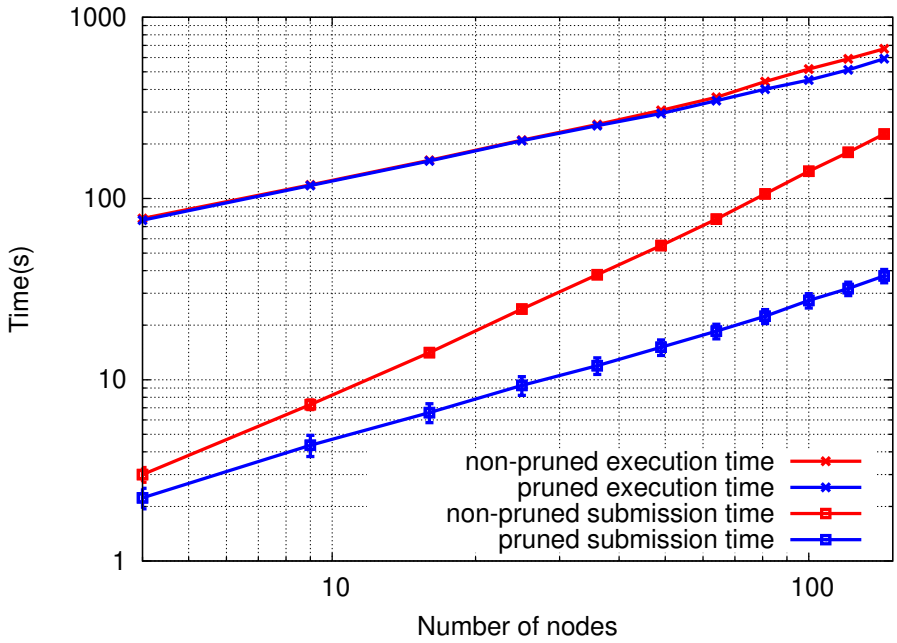


Figure 4.1 – Impact of pruning the task graph on the submission time and the execution time for a Cholesky matrix decomposition kernel from the Chameleon Linear Algebra library on top of STARPU. The test case is a matrix of side size 40,960 for one node; and we keep the same amount of memory per node with increasing numbers of nodes [3].

distributed platform, increases the runtime system processing work on each process, due to the increasing number of tasks being discovered over the number of tasks actually executed by each process. However, one may observe that a full knowledge of every task in the graph is not actually needed to enforce a sequential consistency. A given process only needs to discover the tasks it will execute, e.g. the tasks local to that process, plus the subset of the remote tasks that have local dependent tasks (either input or output dependences) —thus only those tasks that the process will execute and the tasks for which it will generate an incoming or outgoing communication request—. If the applicative code mainly submits tasks with few dependences in regard to the number of participating processes, the cost of processing the local view of the task graph in each process remains moderate and scalable; if the tasks being submitted instead have a large number of dependences, related in some ways with the session size (such as a collective communication operation), scalability may instead be harmed, though only as the result of the applicative code algorithm inherent properties.

This optimization strategy, consisting in keeping only the *relevant* subset of tasks in each process is called *pruning* the DAG. While STARPU performs this pruning step automatically, the benefit is actually higher when —optionally— involving the applicative code in the optimization process, so as to not even submit the unnecessary tasks in each process. Such an optional approach enables skipping the costs of allocating unneeded task structures and of determining the tasks to be pruned in a generic manner. Fig. 4.1 shows an example of the impact of the *pruning* optimization on a weak scaling test [3]. Without this optimization, the task *submission* processing time on a given node quickly becomes prohibitive as the case grows with the number of nodes in the session.

The DUCTTEIP distributed model [175] built on the SUPERGLUE task-based model runtime system [166] offers a distinct take to the issue of extending a single-node task paradigm to a distributed task paradigm. While STARPU relies on tightly coupled distributed dependences (one task on the sending side and one task on the receiving side), DUCTTEIP instead uses a loosely coupled model of distributed dependences, by taking advantage of the data versioning approach of the SUPERGLUE runtime. With this approach, a task on some node produces a given version of a given piece of data, while some other tasks express some dependence on that specific version of that piece of data. This model offers a higher flexibility than STARPU's version-less sequential task-flow, in that it enables expressing graphs of tasks that are not strictly DAGs —such as reductions, for instance— without resorting to artifices. However, the use of a data versioning scheme comes with the counterpart that the pruning optimization described here for STARPU is not directly applicable to such scheme. The reason is that a process using data versioning needs to be aware of the full history of a piece of data to compute the data version number required for a given task, even if this is the only task to ever work on that piece of data in that process. Since this may lead to scalability issues, the DUCTTEIP model is structured in levels, and only the highest level tasks are handled by the distributed memory management. When such a high level task is ready to run, the execution is passed on to low level shared-memory tasks handled by the local instance of the SUPERGLUE runtime. It is however up to the applicative code programmer to organize its distribution accordingly and partition it into relevant high-level and low-level tasks.

The OCR [117] (the *Open Community Runtime*) distributed task-based model specification relies on the use of globally unique identifiers to designate pieces of data. Such data blocks are said 'relocatable' in that they can be moved among nodes and replicated; they are accessed through acquire/release semantics. Task parallelism is expressed as event-driven tasks, and tasks can also be moved to other nodes. Since OCR is a specification, it leaves many aspects to be decided by implementers. In particular, the consistency model is not entirely defined in the specification [67]. Moreover, the reference implementation involves tasks blocking while waiting for remote operations completion, which even though other tasks can be scheduled meanwhile, makes it difficult to apply performance modeling techniques on discontinuous tasks. The deferred execution alternative [67] instead relies on an immediate confirmation protocol, involving confirmation messages of remote message processing to be waited for by the worker thread in charge of the execution, thus adding explicit network synchronizations. Whether the pruning technique is applicable depends in part on details left on implementations, such as the definition of how globally unique identifiers are built. For instance, some of these identifiers can contain a sequence number component to specify a data block version. In that case, for participating OCR instances on each node to compute a data block sequence number would also require them to give up pruning or add extra messaging.

Cache Management This idea of involving the applicative code in the optimization process is pervasive in the STARPU distributed computing design. In particular, it is also employed within the management of the cache mechanism implemented to eliminate redundant inter-process data transfers. Since data transfers are expensive compared to computations, STARPU keeps track —in the *submit* frame of reference— of past data transfers and does not actually generate a communication request for a piece of data for which a local copy exists, unless that copy has been invalidated through the sequential consistency process. Through the

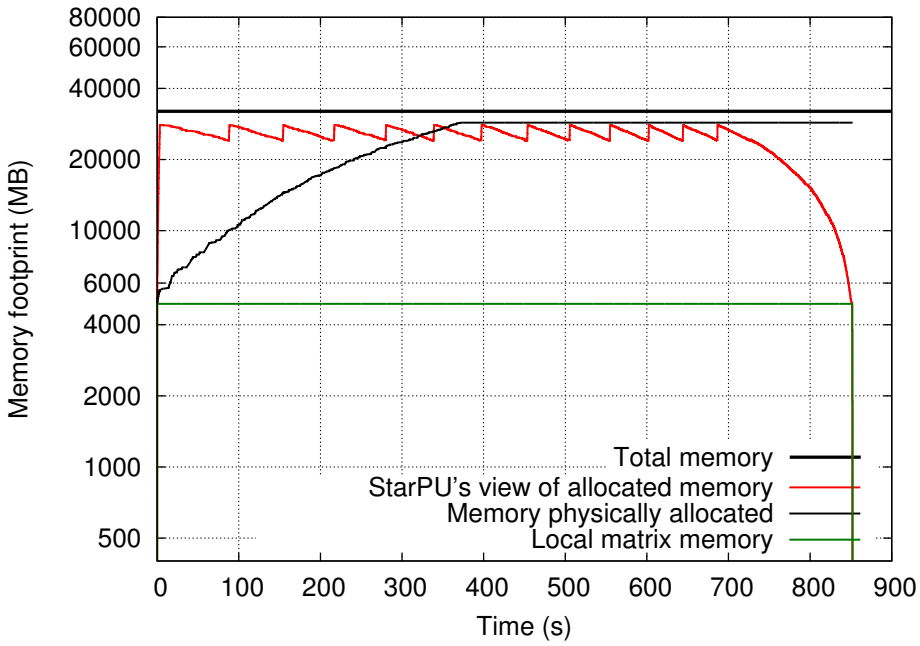


Figure 4.2 – Task submission flow control driven by committed memory threshold [3].

dynamically discovered sequential task flow, a STARPU process cannot determine whether a piece of cached data will never be referred to again in the future; therefore, all valid pieces of data are kept in cache, which may consume memory unnecessarily. The applicative code may help by explicitly stating whether a piece of data will not be referenced again, or by requesting that piece of data to just be evicted for the time being. For the sequential consistency to be respected, the eviction request is inserted at the "current" point in the *submit* frame of reference. If a new reference to the piece of data subsequently arises at some point in the future of this *submit* frame of reference, a new communication request will be triggered.

Task Submission Flow Control The applicative code may further be involved in optimizing the submission flow itself. A sequence of successive task submissions can be seen as a sliding window, with newly submitted tasks entering on the one side of the window and completed tasks exiting on the other side. A large sliding window gives a deep lookahead to the STARPU execution engine, for instance enabling the allocation of incoming communication buffers well in advance to increase opportunities for data transfers to be successfully overlapped with computation, and reducing idleness risks through anticipative planning. Such a deep lookahead can however be unnecessarily expensive, in terms of memory usage, due to the allocation of task structures, data buffers and scheduler queuing resources, and potentially in terms of processing as well due to increased scheduling costs.

Thus, there is a trade-off to find in practice on the length of the lookahead to balance the performance benefits and the resulting overhead. For that, the applicative code may define a pair of `stop`/`resume` conditions on the task submission flow: when the `stop` condition is triggered, for instance as the number of current tasks in `submitted` state reaches some upper threshold, STARPU's task submission function blocks until the `resume` condition is met in its turn, such as the remaining number of tasks in `submitted` state falling down below

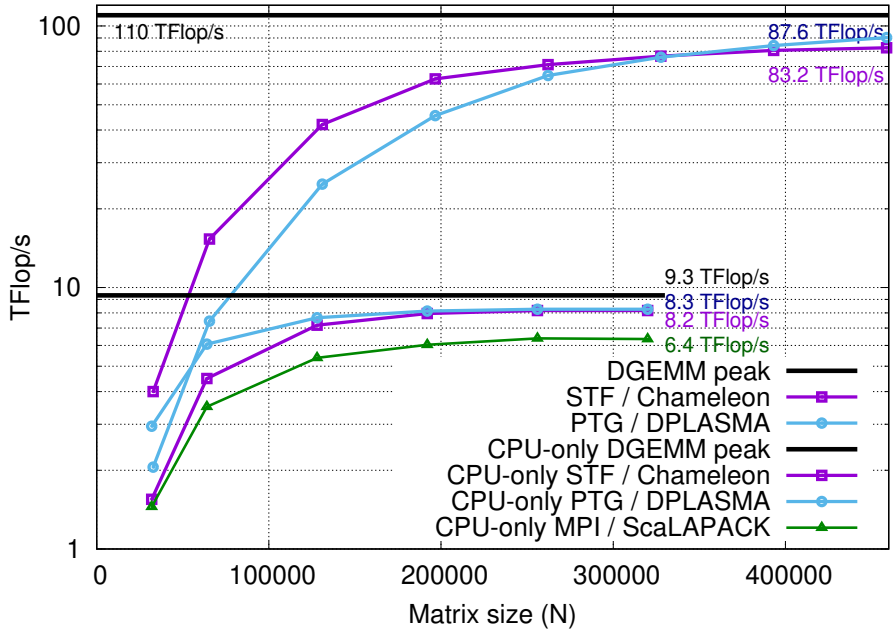


Figure 4.3 – Performance results for a Cholesky decomposition kernel from the Chameleon Linear Algebra library on top of StarPU on 144 nodes of the TERA-100 cluster (1152 CPU cores and 288 GPUs) and comparison with alternative libraries [3]. The Y-axis is a logarithmic scale.

some lower threshold. Other conditions may used instead, such as the quantity of memory currently committed for executing the tasks in the lookahead window. Fig. 4.2 shows the effect of controlling the flow of task submissions using condition thresholds on the amount of memory committed at any time, for a Cholesky decomposition test case [3]. Successive task submissions gradually increase the amount of committed memory, while tasks reaching completion decrease the amount of committed memory. The seesaw-like plot on the figure shows the instantaneous committed memory amount from the StarPU point of view. The top of the seesaw "teeth" corresponds to the upper threshold, beyond which the task submission flow is blocked to prevent a memory overflow, while the bottom end of the teeth corresponds to the point where task submission is resumed.

Results Figure 4.3 shows the performance obtained on the TERA-100 cluster, both in the homogeneous (only CPUs being used) and the heterogeneous (all computational units being used) cases, plus a comparison of these results with alternative libraries: ScaLapack [29] as the MPI-only reference point, and DPLASMA [34], a linear algebra library built on top of the PaRSEC runtime system using a compiled, parameterized task graph (PTG) approach. The experiment, platform and settings are detailed in [3]. The main lesson learned from the experiment is that a fully distributed implementation of the sequential task flow model with dynamic task discovery, as offered by the StarPU runtime system, is competitive against a compiled PTG approach, provided the optimizations discussed above are enabled. Even though the PTG approach enables the runtime system to work on a compact representation of the entire graph of tasks, the fully distributed STF approach with its local decision making

property and an appropriate lookahead depth brings a similar advantage, while preserving the natural, sequential layout of the algorithm, and avoiding the development cost of expressing this algorithm in the artificial formalism of a PTG domain specific language. Incidentally, a flavor of the Sequential Task Flow paradigm was subsequently implemented in PARSEC as well, under the name *Dynamic Task Discovery* [95] (DTD).

4.3 Discussion

Our study on associating a task-based computing runtime system and a communication runtime system under a unique programming model confirmed the virtues of synergy that could be expected from such an association. Moreover, this synergy comes at a moderate cost in terms of development, that is, for the applicative code to supply an initial data distribution. The dynamically discovered data dependences in the unfolding of the sequential task flow are then sufficient to infer the inter-node transfer requests to be made at the level of the communication runtime system, and even to transparently overlap such transfers with computation, provided enough parallelism is available. Moreover, we saw above that these good properties translated into competitive performance, compared to the state of the art. Many aspects can then be discussed from this general overview.

4.3.1 Granularity and Partitioning

Some of the discussions topics naturally overlap with the discussion about computing runtime systems (see Section 3.5). Among these topics, the issue of the granularity (e.g. the computational weight) of tasks is a prominent one here also. It is most often tightly related to the issue of data partitioning, with the option of creating more, shorter tasks working on smaller pieces of data (smaller matrix tiles, for instance), or creating less, longer tasks working on coarser pieces of data. The granularity of tasks impacts the load-balancing and the scheduling management cost as for computing runtime systems. The granularity of the data partitioning impacts the cost of inter-node data dependences in terms of both latency and bandwidth, as well as the size of communication buffers, and the ability to adjust the resulting inter-node load balance. In some cases, such as stencil-like computation kernels, for instance, partitioning into a finer grain also brings the overhead of a larger set of *ghost cells* to handle at the boundaries of the resulting partition pieces. As noted in the computing runtime system discussion, there is not much available at this time in terms of automatic granularity selection, and the data partitioning issue makes the design of generic, automatic approaches even more complex, due to the interplay of these numerous factors. Our work on TAGGRE (see Section 8) is not applicable here since it focuses on aggregating overly short tasks into more profitable coarse tasks from a scheduling and management point of view. In this context, an approach such as the Dependent Partitioning Language [111] (DPL) from Stanford's Legion Project, associating partitioning automatisms with the cooperation of the applicative code programmer —through a constraint declaration Domain Specific Language (DSL)— may be the way to go. One the one side, the language makes it easier for the programmer to explore multiple strategies; on the other side, the programmer helps the automatic partitioner by making the problem more tractable and avoiding heuristics pitfalls.

4.3.2 Scalability

Scalability is also, expectedly, a major issue for distributed computing. We reported above on the scalability properties of the Sequential Task Flow (STF) paradigm associated to a task graph pruning optimization stage, on each node, to trim edges and vertices irrelevant to that node. This is true for an applicative code as long as none of its tasks has a number of dependences correlated to the number of nodes in the session. Otherwise, the pruning stage may not be effective at large scale. Moreover, due to the dynamic discovery of the task flow, the collective data transfers resulting from such dependences have to be detected and optimized on the fly, as the task flow unfolds [63]. Also, if the rate of tasks passing the pruning screen varies a lot from a node to another, there is a risk that the progression of the participating processes may diverge a lot within the "virtual" global task graph, potentially causing some processes to stall not only because some incoming dependence is not yet resolved, but also because a buffer has not yet been allocated on a remote process to accept some outgoing dependence, owing to the fact that the remote process has not yet discovered that dependence —due to the longer processing of that node's flow, or due to the triggering of the task submission control flow on that node—. An increasing number of processes may exacerbate this issue, depending on the dependence pattern of the applicative code and on the way the workload is distributed among the processes, while making it difficult to detect when such stalling conditions arise. Besides, the same characteristics of local, unsynchronized decision making that is the basis of the fully distributed STF efficiency is also complexifying the debugging task. For instance, an erroneously pruned task may go undetected and wreak havoc on the applicative code correctness. Thus, it is likely that large distributed task-based applications will leverage a combination of the three main model variants: The sequential task flow especially for dynamic and irregular phases, a compiled approach such as the parameterized task graph for regular phases, and a nested master-slave approach [40] or version-based approach [175] to organize the application into a hierarchical, but hopefully more programmer-tractable architecture.

4.3.3 Load-Balancing and Fault Tolerance

Two major topics have been left aside in our work on distributed computing so far, namely the issue of load balancing and the issue of fault tolerance. Even though they are distinct, they share many aspects, in terms of necessitating a reaction to an evolving distributed situation, as some nodes become overloaded, underused, or become unavailable due to some fault; in terms of necessitating a trade-off on the preventive measures between their effectiveness and their overhead; and also due to the fact that the recovery process after a fault may require a re-balancing on the workload if the amount of computing resource available has been reduced as the result of the fault. The reason for leaving these topics aside until now has mainly been a matter of prioritizing challenges.

Fault Tolerance The choice of assigning these two challenges a lower priority may appear surprising, especially for the case of fault tolerance in a context of high performance computing. Indeed, the observation of the shrinking projected Mean Time Between Failures (MTBF) due to the densification of HPC computing platforms —e.g. a low probability of failure event per unit, that is, on a microscopic scale, being highly probable for the computing platform as a whole, that is, on a macroscopic scale— has been increasingly emphasized over the last

two or three decades. For instance, early fault-tolerance enabled implementations of MPI such as FT-MPI [75] or MPICH-V [33] have been proposed about twenty years ago now, and the abstract of the FT-MPI paper (from 2000) reads "*As current HPC systems increase in size with higher potential levels of individual node failure, the need rises for new fault tolerant systems to be developed.*". Yet, while discussing with actual users of our Team's software in general and of StarPU in particular, we did not receive highly pressing requests from these users for specific fault tolerance support over other features, though this is likely to change in the future. Also, even though the two fault tolerance-enabled MPI implementations cited above are two decades old, fault tolerance mechanisms are still not standardized in the MPI Specification, at the time of this writing. However, on-going work at the level of the MPI-Forum seems to hint that such a fault tolerance addition to the MPI Specification may be in the coming; and the advent of exascale-era supercomputers is likely not entirely unrelated to that evolution. On the StarPU side also, a PhD thesis study by Romain Lion started on October 2018 on the topic of fault tolerance in StarPU, under the supervision of Samuel Thibault.

Load-Balancing The distributed load balancing aspect in StarPU has, until now, been entirely left upon the applicative code programmer, who is responsible for giving the initial data distribution, and for altering data ownership afterwards as needed to induce load re-balancing. The runtime system has now reached a satisfying level of maturity on its core set of functionalities, to consider integrating some automated mechanisms in it, first for detecting situations of distributed imbalance, and then for proposing load re-balancing counter measures. There is an abundant literature on the theoretical aspects of load balancing for distributed computing. StarPU's STF programming model brings both challenges and opportunities in integrating algorithms from the theoretical corpus. On the challenges side, the requirement for scalability —thus for favoring local interactions and avoiding global barriers— invites a preference for local diffusions algorithms [144], with the associated risk of non-optimality from the global point of view, and therefore of pathological cases if the workload is evolving quickly on the global scale. On the opportunities side, however, a data redistribution in the StarPU sense is merely a redistribution of data ownership at some point in the task submission frame of reference: this means that the redistribution does not by itself trigger data movements on the network; instead, such movements will gradually be performed if/when needed, to serve subsequent data dependences. Moreover, the load re-assignment needs only to be seen on every node at the same *logical* point if the STF, leaving some flexibility on how and when to compute re-balancing measures. And, the load balancing algorithms may take advantage of both the task graph, for some lookahead insight, and StarPU's performance modeling engine to help establish relationships between the data distribution and the resulting workload distribution.

4.4 Conclusion on Performance Portability of Distributed Computing

In this chapter, we looked at combining a computation runtime system and a communication runtime system to build an unifying and synergistic parallel+distributed model based on the sequential task flow paradigm, as the result of the PhD work of Marc Sergent. We saw how this approach associated with a fully distributed execution model enables decisions to be taken locally, and inter-node interactions to be limited to communication with nodes'

immediate neighbors in the DAG, two key properties for scalability. We also considered how engaging the applicative code programmer in the optimization process made it possible to dramatically trim down the amount of processing required in terms of runtime system execution management. We then discussed how to go further from this robust foundation, to optimize execution and to foster scalability even more; and also to add increasingly necessary supports for fault tolerance and load balancing.

This chapter also concludes the first part of this manuscript, in which we have examined the key role of runtime systems to support performance portability in HPC applicative codes, by abstracting hardware-specific details from programmers while optimizing the processing of work requests. From the experience gathered in working on several runtime systems for communication and for computation as part of my research work, we have studied the design choices of such runtime systems regarding the programming models, execution models and architectural layouts, and we have discussed how these choices impact aspects such as the runtime capabilities, or the range of possible optimizations. The second part of this manuscript will now present several works related to increasing the programmability of HPC applications.

Part II

Approaches to Programmability Enhancement

Table of Contents

While runtime systems can help in the programming task by acting as delegates in performance portability management and in providing for abstracted programming interfaces, building entire high performance computing applications involves countless other aspects such as writing the necessary pieces of code and assembling them together. Many such aspects can be tailored for the special needs of HPC; this second part presents a selection of these aspects and the works we have conducted to improve their status in the context of HPC.

The increasingly high level of parallelism available in computing hardware (see Chapter 3) necessitates to find a correspondingly ever increasing level of parallelism in software to take advantage of it. Some of this software parallelism can be automatically detected and exploited by processors through pipelining, instruction reordering and superscalar execution. Another part of this software parallelism can be extracted by compilers through automatic vectorization and parallelisation mechanisms. To go beyond, however, some involvement of the programmer becomes necessary. In that situation, language design can help in making the parallelization job more tractable for the programmer, and possibly more efficient in terms of resulting performances. The works we have conducted in the context of the OpenMP language [2] join this global effort to make parallel programming more effective (see Chapter 5).

Due to the cost of optimizing applicative codes, the incentive to reuse already optimized pieces of code is high in computer programming in general and in HPC in particular. Doing so, however, necessitates to make sure that *composing* multiple software elements optimized for HPC actually results in an efficient aggregate as a whole. While the notion of software components has existed for a long time now [159, 158], the context of HPC brings specific problems such as arbitrating access to computing units, hardware resources and peripherals between such software components, and making sure that such software elements can interoperate. We have been working at several levels to address issues of interoperability and composition of codes in HPC (see Chapter 6).

Moreover, optimizing a piece of code for a given architecture —by implementing techniques such as iterations blocking, for instance— may alter the layout of the underlying algorithm to the point where it becomes difficult to understand, to maintain and to port onto other architectures. The work we have conducted on the INKS framework [71, 72] builds on the concepts of domain specific languages and of aspect-oriented programming [42, 106] to enforce the notion of separation of concerns, where an algorithm and a set of optimization strategies to be applied to it are written separately, and then processed together by a compiler to produce an optimized executable instance for a given platform (see Chapter 7).

Chapter 5

Enhancing Programmability through Language

In this chapter, we present works conducted on the topic of high-level parallelism expression through the OPENMP language, especially on the design and development of the KSTAR OPENMP compiler. While the runtime systems presented in Part I bring many benefits in terms of performance portability, and therefore relieve programmers from some painful optimization work, the programming interfaces offered by such runtimes usually are too technical to be easily learned and mastered by non specialists. Specially crafted programming languages offer a means to lower the "usability bar", by leveraging the power of a compiler to handle the low level details for a relevant subset of usage patterns. The challenge is then to do so while preserving most of the runtime system's benefits.

5.1 Context

Task-based runtime systems such as STARPU [13] each come with their own specific application programming interfaces (APIs). While many similarities exist among these APIs in terms of general concepts, the differences in paradigms, expressiveness and implementation details are substantial enough to make it difficult to port an application written for one task-based runtime system onto another task-based runtime system. The OPENMP language is a *de facto* standard for parallel programming in HPC maintained by the OPENMP Architecture Review Board (OPENMP ARB). While initially focused on the thread-based, fork-join parallel programming paradigm, it was extended to support basic, independent task parallelism as part of its specification's 3.0 release, and extended further with dependent tasks in the 4.0 release. Within the context of the KSTAR INRIA development action grant, and together with developers of the task-based runtime KAAPI [82], we investigated the possibility to use the OPENMP language as a portable, generic interface to STARPU and KAAPI. This effort led to the development of a C/C++ source-to-source compiler, KLANG, later renamed into KSTAR, to translate OPENMP annotation directives into calls to the selected runtime system, either STARPU or KAAPI.

5.2 The KSTAR OPENMP Compiler

The KSTAR compiler is based on the LLVM framework and on the CLANG-OMP front-end. Initially developed by INTEL as a distinct project from LLVM/CLANG, the CLANG-OMP OPENMP front-end has since been merged into the main LLVM/CLANG repository. KSTAR translates OPENMP directives into calls to task-based runtime system APIs. In terms of programming models, it supports legacy fork-join OPENMP constructs such as `parallel` regions, `parallel for` loops and sections. It also supports independent tasks as defined by the OPENMP specification revision 3.1 as well as dependent tasks introduced with OPENMP 4.0.

The design choice of building a source-to-source compiler instead of a source-to-binary one comes from the intent to leave the application programmer the freedom to select the most suited binary compiler (GNU GCC, LLVM/CLANG, INTEL ICC, ...), and to focus our work on runtime-related code generation instead of low-level optimizations for which existing compilers already excel.

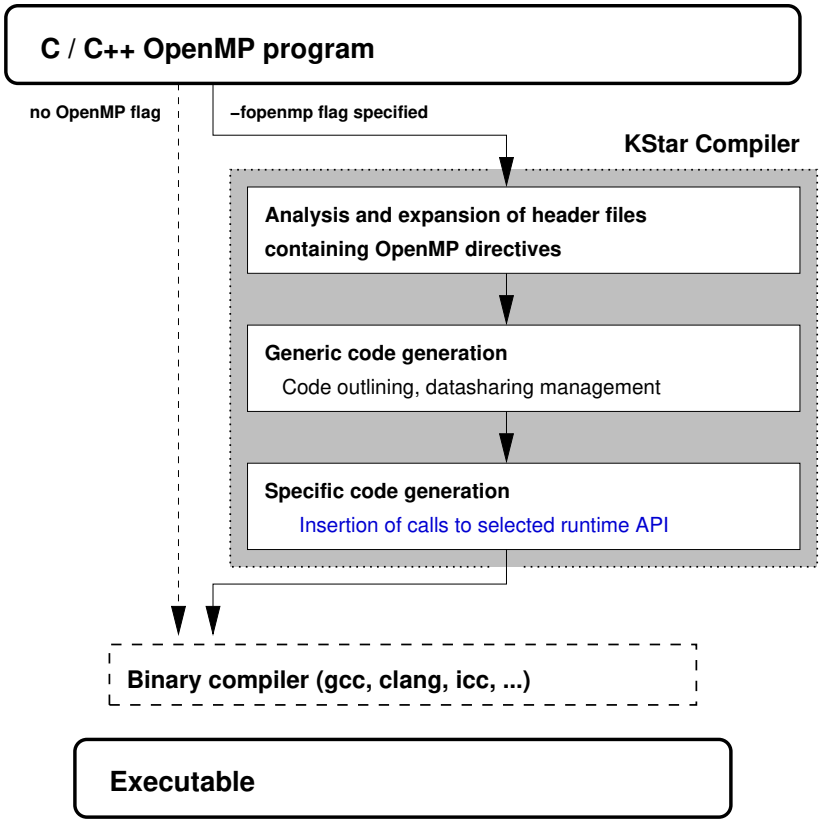


Figure 5.1 – Organization of the KSTAR compiler.

A description of the compiler organization is given on Figure 5.1. The compiler is based on the technique called "abstract syntax tree (AST) *pretty printing*": the initial, OpenMP-annotated, source code is parsed by the CLANG-OMP front end, resulting in the AST representation. The AST then goes through a pretty printing phase, in which AST nodes are

converted back to a textual representation. It is during that phase that OpenMP constructs are substituted with their back-end runtime system counterparts. C and C++ constructs not involved in —and not directly referenced from— OpenMP constructs are output back unmodified (except possibly for indentation). The alternative technique consisting in transforming nodes of the AST tree directly, as done in the GNU GCC Compiler Suite OpenMP front-end, for instance, was not technically possible with LLVM/CLANG's AST at the time the KSTAR compiler was designed, and it still is not possible at the time of writing this manuscript.

KSTAR first analyzes included files and unfolds the ones containing OPENMP pragmas, so as to collect every OPENMP directive in a single source file. After this step, KSTAR expands the OPENMP directives, performing related code outlining and data-sharing transformations as needed. The code outlining step involves extracting the source code bodies of OPENMP block directives such as parallel regions and tasks, and wrapping these pieces of source code into individual functions that can then be called by the back-end runtime system. The code outlining phase must manage data sharing aspects as well, since the outlined code must still be able to access the data items it references, and the modalities of accessing each data item must follow the OpenMP data sharing clauses specified by the programmer or the default behaviour defined in the OpenMP specification document [125]. Finally, KSTAR generates the runtime-dependent source code to issue calls to the selected runtime API.

5.3 The OpenMP Runtime Support in StarPU

Most of the work of the STARPU backend of KSTAR is to generate code to prepare data structures and call STARPU routines, where all the actual logic is implemented. However, the native STARPU programming, as described in Section 3.4, differs in some important ways from the OPENMP programming model. As a result, the STARPU backend of KSTAR actually targets a software layer —named STARPU OPENMP *Runtime Support* (SORS)— implemented mostly on top of STARPU and for a small part within STARPU. The role of this layer is to extend the native STARPU API with the necessary semantics and routines to implement compliance with the OPENMP specification [125].

Fundamentally, the specification defines two kinds of tasks, on which the OPENMP model is built: *implicit tasks* that support the "legacy" fork-join programming model of OPENMP and are created by the `omp parallel` pragma —one implicit task is created for each participating thread of a parallel region— and *explicit tasks* created using the `omp task` pragma introduced as part of OPENMP 3.0 and extended with data dependences in OPENMP 4.0. The particularity of these tasks is that they can block, they can create new nested tasks, and they can wait for their nested tasks' completion, while these properties are not authorized in the native STARPU model.

The SORS layer implements these capability within the STARPU model by utilizing UNIX SYSTEM V *contexts* to integrate the concept of *continuations* [134] in OPENMP-enabled STARPU tasks —an implementation on top of a light-weight threads library would also be feasible [50]—. Using contexts mechanisms supplied by the `makecontext` and associated `libc` routines, the SORS layer assigns a dedicated stack to each OPENMP tasks. When the STARPU engine executes such a task, the task immediately switches from the STARPU's worker thread stack to its own dedicated stack. It then proceeds with execution until it completes or until it blocks. At this point, it switches back to worker thread stack, while saving its context if the reason for the switch is a blocking situation. The STARPU task carrying the OPENMP task

either definitively completes as a regular task, or "temporarily completes" if on a blocking condition. In the later case, the StarPU task becomes a continuation waiting for some wake-up condition(s) to be met. Until these conditions are met, the StarPU task is not considered for scheduling, similarly to tasks waiting for dependences to be fulfilled. Once the condition for wake up are met, the task is again pushed to the scheduling engine, and assigned for execution on some worker thread. When the task eventually gets executed, its OpenMP context is restored on top of the OpenMP task stack and the execution proceeds from where it left before blocking. Other OpenMP constructs are then built on top of these blocking tasks.

The extension is transparent to most of StarPU mechanisms, and in particular to scheduling policies. As a consequence, StarPU scheduling policies are available with the SORS layer as well. Moreover, OpenMP-enabled tasks can cohabit with regular, non-blocking StarPU tasks. Besides, while the OpenMP programming model is relaxed, compared to StarPU's native implementation of the *sequential task flow* paradigm (STF), the OpenMP model is still sufficiently constrained to enable dead-lock free execution independently of the scheduling policies. A key characteristic of the OpenMP model, for that, is the fact that task dependences are only enforced between sibling tasks: thus each nesting level can be seen as defining one self-contained nested STF space.

The OpenMP execution model also specifies that the *main* thread (called *master* thread) does actually act as a *worker* thread: it takes an active part in executing tasks and is bound to a core, while in the native StarPU execution model, the task submission thread belongs to the applicative code (which may or may not bind it) and does not participate in task execution (see Section 3.4.2). Thus, StarPU adapts its main thread policy for OpenMP compliance when used through the KStar compiler. Due to this different master thread involvement between StarPU's "OpenMP" mode and the native StarPU mode, slight differences in behaviour and performance results may be observed between a native StarPU program, on the one side, and its OpenMP algorithmic equivalent compiled with KStar and run with StarPU in "OpenMP" mode, on the other side.

5.4 The Commutative Dependence Type Extension

We used KStar and StarPU to explore OpenMP extensions, to take advantage of the possibility to work both on the compiler front-end side at the language and programming model level, and on the back-end, runtime system side on the execution model. We present an example of such an extension below, to support commutative dependence types. A more detailed presentation is given in [2].

In the *sequential task flow* parallel programming model, the order in which the tasks are created can be extremely sensitive for some kind of applications, leading to different instantiations of the directed acyclic graph (DAG) of tasks. The fundamental reason is that the `inout` dependence type, specified for data elements being updated by an OpenMP task, indeed not only forces mutual exclusion of tasks with respect to that dependence data, furthermore it forces sequential consistency as well. For instance, if a task T_1 and a task T_2 are submitted in that order to update a piece of data A, and if T_2 becomes ready to run before T_1, T_2 will nevertheless *not* be allowed to start before T_1 is completed. This results in tasks accessing that piece of data being forcibly executed in the task creation order. For many algorithms, this is the expected behaviour. The Fast Multiple Method (FMM) implemented in the ScalFMM library [4] to speed-up N-body algorithms is a typical example where this

strong ordering enforced by the `inout` dependence type is detrimental. The FMM method is a hierarchical algorithm introduced in [87]. Its principle is to approximate the far-field —that is, the interactions between far-apart particles— while maintaining a desired accuracy, exploiting the property that the underlying mathematical kernel decays with the distance between particles. While the interactions between close particles still remain computed with a direct Particle-to-Particle (P2P) method, the far-field is processed using a tree-based algorithm instead. A recursive subdivision of the space is performed in a preprocessing symbolic step.

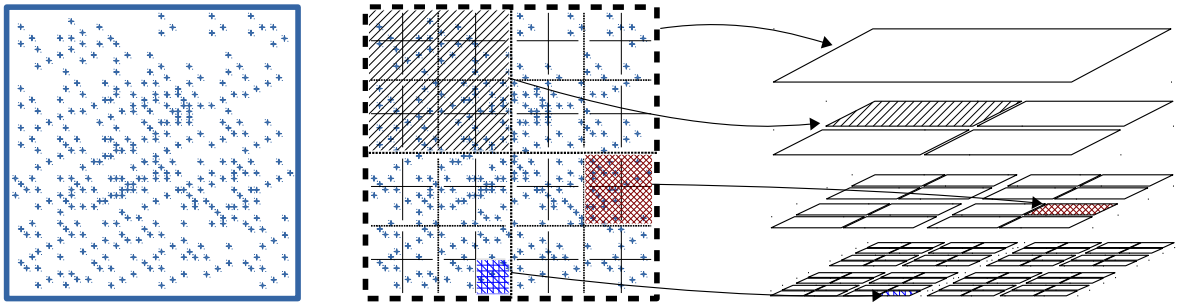


Figure 5.2 – 2D space decomposition (Quadtree). Grid view and hierarchical view.

This recursive subdivision is usually represented with a hierarchical tree data structure, and we call the height of the tree h the number of recursions. The actual type (quadtree, octree, ...) of the tree is selected according to the dimension of the problem. Figure 5.2 is an example of an octree showing the relationship between the spatial decomposition and the data structure, and we see that each cell represents its descendants composed of its children and sub-children. The multipole (M) of a given cell represents the contribution of its descendants. On the other hand, the local part (L) of a cell c represents some contributions that will be applied to the descendants of c. These local contributions in a cell c come from the potential of particles/cells that are not included by c. Forcing a strong ordering for accumulating incoming contributions on a given cell or particle therefore overconstrains the scheduling problems, eliminating a number of schedules that would yield valid results from the FMM algorithm point of view. The accumulation of incoming contributions is inherently commutative, thus forcing an arbitrary execution order may unnecessarily delay the execution of ready tasks, wasting parallelism.

To address this issue, we proposed to introduce a new OpenMP data dependence type, to express commutative update operations [2]. The commutative update type still enforces mutual exclusion on the piece of data being updated, but relaxes the ordering of the incoming updates. From the application point of view, this merely involves changing the task creation step to use the new dependence type. The introduction of this new data dependence type is necessary, because there is no satisfactory alternative with pre-existing OpenMP constructs:

— `omp critical`: The construct introduces a mutually exclusive critical section that only one thread may access at any time. It can be used to perform exclusive processing in "no specified order", as required. However, it breaks the data dependence flow between tasks. Moreover, all *anonymous* critical sections share the same exclusion lock, making it too broad to use, if more than one piece of data is concerned by this update scheme —FMM computations typical datasets can reach on the order of 10^6 to 10^9 elements—. Named critical regions do not really help here: they cannot depend on a dataset discovered at run-time, because the names have to be known at compile-

```
#pragma omp task depend(inout:Z) // Task T1
 ...
#pragma omp task depend(mutexinoutset:Z) // Task T2
 ...
#pragma omp task depend(mutexinoutset:Z) // Task T3
 ...
#pragma omp task depend(inout:Z) // Task T4
 ...
```

Figure 5.3 – Example of `mutexinoutset` dependence type usage.

time. Also, expressing a task updating several pieces of data at the same time in a commutative way would require nesting multiple critical sections, which is awkward to achieve while preventing deadlocks.

— `omp atomic`: This construct can be used to implement locks, thus critical sections as well [108]. However, while this construct does not suffer from the compile-time definition requirement of the `omp critical` sections, it suffers from all the other same drawbacks. Moreover, it causes busy waiting, while `omp critical` typically implements passive waiting.

— Parallel reductions: The OpenMP specification offers multiple means to perform parallel reductions, within parallel regions, or between tasks. However, parallel reductions are not well suited for the considered use-case. Indeed, the purpose of parallel reductions is to increase the amount of available parallelism using some *ad-hoc* support from the runtime system: the runtime trades increased memory consumption for increased parallelism. In the considered use-case, however, the amount of parallelism is actually *very* large. Thus, trading memory consumption for more parallelism would be a net loss.

In consequence, we designed an OpenMP extension defining a new data dependence type named `mutexinoutset` (after having initially been called `commute`, referring to the commutativity it expresses), delineating a mutually exclusive input/output set of tasks. Such a set can be seen as defining a relation of equivalence between all the tasks in the set, with respect to the piece of data referenced in the dependence type clause, which can be executed in any order with respect to each other, but not at the same time. The set as a whole is itself ordered within the data dependence flow, as if it was a single task with an `inout` dependence type clause on the same piece of data.

Listing 5.3 shows an example using the `mutexinoutset` dependence type. The tasks declared in the listing will execute as follows: Task `T1` will execute first due to the strict sequential ordering enforced by the `inout` dependence type; Task `T2` and Task `T3` will execute next, in any order with respect to each other due to the `mutexinoutset` dependence type, but not at the same time since such dependence type still enforces mutual exclusion; eventually, Task `T4` will execute last, due to the strict ordering of the `inout` dependence type with respect to the `{T2,T3}` set of tasks.

Figure 5.4 shows a comparative example using the ScalFMM library through its port on OpenMP dependent tasks. Three variants are tested: a direct build of ScalFMM with the GCC compiler and run with GCC's OpenMP runtime system (e.g. `libgomp`); a build with the KStar compiler using the StarPU backend and regular `inout` dependence types (labelled as "`starpu/openmp/commute`" on the plot); and a build also with KStar using the StarPU backend, but using the `mutexinoutset` dependence type (labelled as "`starpu/openmp/commute`")

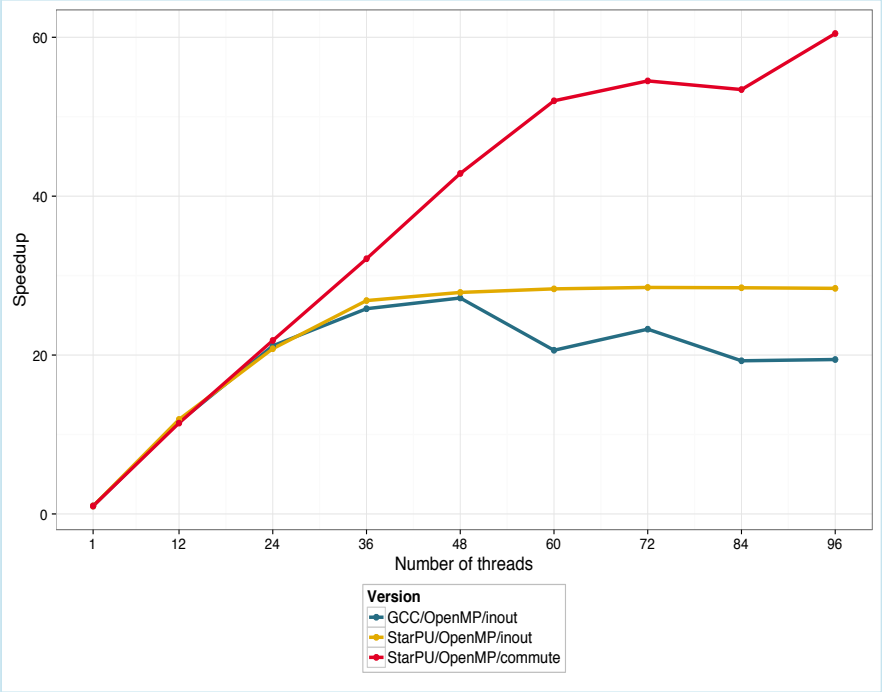


Figure 5.4 – Comparative speedup experiment with a SCALFMM test-case: 10^8 particles Ellipsoid distribution, depth 11 octree. 96-core SGI UV2000 platform. Speedups computed against the GCC version result using 1 thread.

to specify dependences on commutative incoming contributions. The test-case uses a large ellipsoid-shaped distribution of particles, and is run on a 96-core SGI UV2000 NUMA machine. While the KSTAR + STARPU version using `inout` performs better than the GCC version, it still struggles to feed the 96 worker threads to fully utilize the machine, due to the amount of parallelism wasted by the over-constrained sequential ordering of the `inout` dependence type. Instead, the version using the `mutex inout set` dependence type to declare commutative task dependences is able to take advantage of that extra parallelism to obtain a much better speedup. As a result, the `mutexinoutset` dependence type extension was eventually integrated in the OPENMP specification, as part of the major 5.0 revision.

Going further, a variant of this dependence type, the "*non exclusive*" inout set (simply spelled `inoutset`) has since been included in the revision 5.1 [126] of the OPENMP specification. It defines an inout set of tasks, similar to a `mutexinoutset` with respect to tasks that do not belong to the set; however, it relaxes the mutual exclusion constraint between tasks withing the set. The rationale is that in some cases, the programmer knows that several tasks can concurrently write to a piece of data without the risk of causing data consistency issues, by design. An example would be a task accessing the lower triangle part of a square matrix, and another task accessing the upper triangle part of that same matrix, without overlapping on the matrix diagonal.

5.5 Discussion

The work on designing an OPENMP layer on top of the STARPU runtime system follows a recurring pattern that we have followed in designing runtime systems, that is, to complement a low level *native* API to a runtime with a higher level implementation of a standard API. For instance, the MARCEL multithreading library developed as part of the PM^2 framework was extended with a POSIX compliant PTHREAD API, and both the MADELEINE and NEWMADELEINE libraries were extended with MPI compatibility, MADELEINE through a port of MPI-CH on top of it [20] and NEWMADELEINE through its own MAD–MPI API [17]. This recurring pattern has several aims. First, naturally, to broaden the scope of each runtime system, by enabling a range of standard compliant applications to readily work on top of it. Second, more importantly, to study the gap between what can be expressed through a standard API constrained by its specification, and a dedicated, unrestrained native API, in order to design new standard extensions. And third —in the case of OPENMP— to explore the relationship between the language, the compiler and the runtime system.

5.5.1 Further Experimentations

On this last aspect, the KSTAR compiler enabled experimenting with language extensions, as shown above. Another complementary direction of experimentation is to explore runtime system mechanisms while keeping the language and the compiler constant. For that, the STARPU SORS layer also exports an interface ABI-compliant with a subset of the LLVM OPENMP runtime system specification. Thanks to these two experimentation possibilities of working at the language level and/or the runtime system level, multiple topics could be investigated.

A first topic, which happens to be popular when we present STARPU followed by KSTAR, is to explore distributed computing from an OPENMP program. As presented in Section 4.2, STARPU supports expressing distributed computing applications from the *Sequential Task Flow* (STF) paradigm, in a virtually identical way to a single node application. Thus, at first sight, a single node OPENMP application compiled with KSTAR into a STARPU application should also readily be usable for distributed computing. Yet, the reality is more complex. While STARPU has a notion of data handle, which can be seen as a kind of "fat pointer", e.g. an indirection to a piece of data with additional metadata and semantics, OPENMP so far does not define a similar indirect way to designate pieces of data involved in computations, beyond regular C pointers. Thus, there is no general way to attach a notion of node ownership to a piece of data, which is the foundation for managing data exchanges in a distributed computing session. Some workarounds are possible such as using an "iso-allocation" scheme [9, 14], as done in CHARM++ and OMPSS, which reserves distinct address ranges of the virtual memory address space to distinct nodes, to ensure that a given C pointer always uniquely designates its node owner. However, such a mechanism comes with its own issues. For instance, it requires every memory area potentially involved in distributed exchanges to be allocated through a specific allocator, leading to some possible interoperability problems if not all of these areas are allocated by the OPENMP code. And more critically, it does not keep any information about data objects' size and layout (e.g. such as the *width*, *height* and *leading dimension* of a matrix tile). Instead, recognizing data as a "first-class citizen" in OPENMP with a dedicated data entity —just like task regions and parallel regions are dedicated work entities— would likely be the way to go, with a straightforward mapping onto STARPU's

data handles.

Together with an indirection on data, an indirection on tasks would also enable scheduling OpenMP applications on heterogeneous, accelerated nodes, using StarPU scheduling policies. Indirections on data would enable transparent transfers between the main memory and the memory space of discrete accelerators in the same way as distinct nodes. Indirections on task implementations would enable the run-time selection of a CPU or accelerator implementation by the scheduling policy. On this aspect, the notion of variants introduced in OpenMP 5.0 will be instrumental in enabling such an indirection on tasks, by letting the application programmer specify multiple alternative implementations, e.g. multiple *variants*.

This leads to a second topic of possible investigation, concerning the scheduling process in itself. Indeed, OpenMP runtime systems typically implement *work stealing* strategies for scheduling, which is fine for homogeneous scheduling. For heterogeneous scheduling, however, a reactive strategy such as work stealing, which attempts to correct imbalances "after the fact", is less interesting than proactive strategies that attempt to readily generate an efficient work mapping by *planning*, such as StarPU's policies inspired by the HEFT scheduling algorithm [167]. These policies take into account the relative efficiency of each task on each kind of computing unit involved and the cost of data transfers between the host memory and an accelerator memory. A work stealing strategy may instead cause overhead due to assigning tasks on units that are less efficient to process them, to generating additional data transfers while correcting the imbalance through work stealing, and some more overhead for transfers triggered too late to allow computation/communication overlap. In order to do proactive planning scheduling with OpenMP, however, the resource needs and execution time of a task must be estimated in advance. Due to the versatility of OpenMP, this may not be possible in the general case. For instance, a task may block for an undetermined amount of time; also, it may spawn tasks recursively or even open a new parallel region, in which cases the initial task launched on a single CPU core ends up potentially using several cores. Thus, being able to use proactive scheduling with OpenMP will likely require to limit the scope of tasks being scheduled that way (such as only allowing `final` tasks), or requesting more information (e.g. specifying at creation time the actual "width" of a task about to launch a parallel region, or parametrizing the cost of complex, blocking or recursive tasks when possible) from the application programmer to let the runtime system engine build effective cost models in more complex cases. In many ways, this relates to the discussion about bubble scheduling in the granularity/TAGGRE Section 8.4.

A third topic is the support for interoperability. This encompasses a number of distinct capabilities discussed in Section 6.2.3.

5.5.2 New Alternatives to OpenMP Directives

The domain of high level language-based parallel programming models is currently ongoing a surge in diversity, offering a widely different landscape than the one at OpenMP's inception.

On the *pragma* annotation-based languages side, such as OpenMP itself, another well known contender is OpenACC. Initially launched in November 2011 as an alternative to the (at the time) up-coming support for accelerator devices in OpenMP, the language as since grown; meanwhile, OpenMP on its side has since integrated support for accelerator devices through the `target` family of constructs, to the point where both languages largely overlap nowadays. Yet, even though the question of merging back OpenACC and OpenMP

is recurring every year at the SUPERCOMPUTING conference's "*Birds of a Feather*" sessions, the merge appears unlikely to happen in a near future due to the existing codebase in each language. Beside OPENMP and OPENACC, the XCALABLEMP pragma-based language also remains popular especially in Japan —where it is developed— due to its unique abilities at distributed computing support. The INKS language we developed indeed leverages the abilities of XCALABLEMP in a dedicated backend, see Chapter 7.

One major trend, now, is the rise of the C++ "mainstream" language as a language becoming mature for HPC applications. In a kind of virtuous circle, this rise is driving the C++ language to integrate even more advanced evolutions at a fast pace. A first evolution is the integration of some notions of parallelism right into the C++ language, in particular with the standardization of the multithreading support, the multithread memory model and the corresponding API (thread objects, synchronization objects, atomic routines) in the C++'s Standard Library in C++11 [154] (some aspects, and especially that multithreaded memory model, have also been integrated in C11 as well). While these capabilities are by themselves rather too low-level to be leveraged directly in most HPC applicative codes, as discussed in Section 3.2.2, they constitute important building blocks, optional but readily available for high level models. As of C++17, the Standard Library even integrates some parallel algorithms in its *Standard Template Library* (STL) part.

Meanwhile, the integration in C++ revisions of increasingly more constructs inspired from functional languages, associated to its strong templating capabilities and its improving type inference support in compilers, is fueling several approaches. For instance, KOKKOS [70] and RAJA [27] are two frameworks, each developed by a national laboratory from the USA —KOKKOS is developed at Sandia National Laboratory and RAJA at Laurence Livermore National Laboratory respectively— using C++ capabilities, and especially templates and lambda functions, to provide HPC programmers with abstract environments for HPC programming. Both enable a single applicative code to target heterogeneous computing nodes, and generate parallel kernels (both are able to target an OPENMP back-end), vectorized kernels or accelerated kernels with little to no code modification, and with a single source code, one of the most important requirements from their host national labs. The SYCL C++-embedded domain-specific language [155] standardized by the KHRONOS GROUP, also in charge of specifying the well known OPENGL and OPENCL languages, also builds on this idea of a single-source code file that can be compiled to target heterogeneous platforms: a special compiler extracts C++ kernels to be offloaded to some accelerator and generates the appropriate device-specific code, while a "regular" C++ compiler can compile the same code to a host-only binary code [69]. The ONEAPI initiative launched by company INTEL [124] defines the DPC++ (data parallel C++) language as an extension to SYCL and C++, and also aims at providing HPC applicative code programmers with a single-source code means to target regular processors, accelerators and FPGA reprogrammable processors. While these environments are powerful, they tend to have a steep learning curve, and expose rather low-level details so as to enable such programmers to remain in control, when custom, in-depth optimizations are necessary. In the lot, KOKKOS seems to get a more successful, wider acceptance, likely due to its slightly higher positioning in the abstraction scale.

Finally, as an alternative to the C/C++/Fortran trio, the JULIA language, natively designed to support parallelism and vectorization —and currently undergoing a fast development process and growing community— is showing increasingly impressive performance results, while maintaining a sustained level of abstraction, readability and general scientific programmer friendliness. The couple PYTHON + NUMPY is also becoming very popular, though

with a distinct approach: all the parallelism is hidden within the NUMPY library and the applicative-code remains largely sequential from the point-of-view of the programmer. The versatility of the PYTHON + NUMPY couple in terms of heterogeneous HPC platform support is however limited compared to the languages cited above, and exploiting hardware resources such as some accelerator boards basically requires using a device-specific "*drop-in*" NUMPY replacement [26].

The OPENMP language itself continues to evolve at an increasing pace, as an answer to the evolving hardware landscape, but also under the influence (and perhaps under the pressure!) of the innovations of these alternative models and approaches. The summary of differences between OPENMP 4.5 and OPENMP 5.0 in the specification "Features History" appendix is several pages long, for instance. Among the features added, the support for the *variant* concept of compile-time traits-based code selection follows the trend of single-source support for heterogeneous computing nodes discussed above; the support for *iterators* extends the capabilities of the language in terms of generic programming; the *loop* construct is inspired by a similar functionality in OPENACC, offering a *descriptive* loop parallelisation mechanism to choose from —besides the existing *prescriptive* loop constructs such as `omp parallel for`— and enabling the compiler to freely select the most appropriate loop implementation when the descriptive construct is used instead of a prescriptive one; some preliminary interoperability support is arriving in OPENMP 5.1 [126] as discussed in Section 6.2.3; etc. The resulting language is therefore becoming comprehensive in terms of versatility and scope, but also, as a possibly inevitable downside, increasingly difficult to master in its entirety.

5.6 Conclusion on Languages and OpenMP

In this chapter, we explored the use the OPENMP parallel programming language as a way to target a computing runtime system such as STARPU with high-level interface — instead of using the more powerful but also more complex native API directly— from the experience of the KSTAR source-to-source compiler that we implemented as part of the INRIA technological development action ADT KSTAR, and also from my participation to the OPENMP ARB standardization body, for which I have been representing INRIA since 2016. We saw how the compiler enables studying the OPENMP language expressiveness, as well as designing language extensions and measuring their effectiveness. We then discussed the perspectives opened by these works, and some important directions to investigate next, such as the interplay of OPENMP with distributed computing as a follow-up to Chapter 4, and the need to give data a higher status within the language than is the case at this time. We also discussed the numerous C++-based alternatives to OPENMP emerging and gaining traction, thanks to the fast strengths and paced evolution of C++. Both directive-based parallel APIs and C++ template-based parallel APIs offer a variety of solutions that significantly enhance programmability over targeting runtime systems directly from the applicative code.

Next chapter will now explore another, complementary way of improving programmability by enabling multiple runtime systems and thus multiple parallel pieces of code to be composed in a cooperative way.

Chapter 6

Enhancing Programmability through Interoperability and Composition

In this chapter, we present two distinct, but tightly related works on the topic of interoperability and composition, as facilitating tools for programming. Indeed, the whole act of programming is about decomposing a complex intended operation all the way down to individual, elementary operations that the target machine can understand, while organizing them into a hierarchy of small nested logical groups, such as instructions, statements, functions, modules, etc., that human beings can understand. The topic studied here is about applying this concept to HPC. The first study deals with designing requirements for communication oriented composable modules to be assembled into such kind of logical groups to build network stacks for the IBIS grid computing environment. The second study deals with defining interoperability requirements to enable entire runtime systems to be assembled within HPC applications in an efficient cooperative way, in the context of the European Project H2020 INTERTWINE.

6.1 Composable Network Stack for Grid Computing

During my 2002–2003 post-doctoral position at Vrije Universiteit in Amsterdam, the Netherlands, in the team of Henri Bal, I was involved in a software project called IBIS. The purpose of this project was to develop an environment for exploiting multiple HPC clusters together within the same application, in a setup called *Grid Computing* [78], a precursor setup of nowadays cloud computing platforms, somehow. Due to the necessity for the resulting application and environment to run on platforms discovered only at run-time, the IBIS project puts a tremendous emphasis on portability. It relies on the JAVA programming language and associated virtual machine to enable the application and the environment to run on the grid clusters assigned at runtime without requiring any rebuild. Within this project, I designed the network stack named NETIBIS [19] for the environment. Though the network technologies have evolved substantially since then (and the IBIS project has since ended), the architecture of the NETIBIS network stack addresses technical issues that are still relevant today.

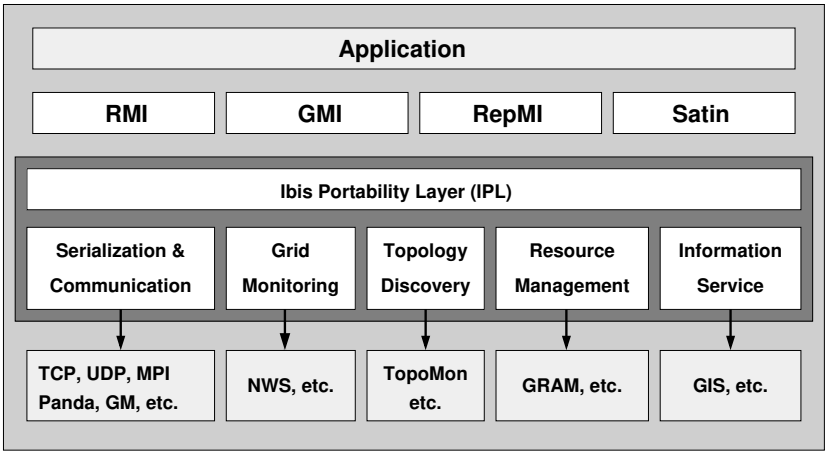


Figure 6.1 – Design of IBIS. The various modules can be loaded dynamically, using run-time class loading.

6.1.1 Premisses and Requirements

IBIS [169] is a JAVA-centric environment for grid programming. IBIS uses JAVA's "write-once, run-everywhere" property to address the intrinsic heterogeneity of grids. Virtually all IBIS communication software and runtime system software are implemented in JAVA, and IBIS runs out-of-the-box on heterogeneous grids. One of the main research problems studied in the IBIS project is how to implement this JAVA-centric approach efficiently.

The structure of the IBIS system is shown in Figure 6.1, and exposed in more details in [169]. The API of IBIS, named IBIS *Portability Layer* (IPL), is a thin interface to several IBIS runtime parts such as *serialization and communication* or *grid monitoring*. Each part can have different implementations —NETIBIS is one implementation of the serialization and communication part, see Section 6.1.2— that can be selected and loaded into the application *at run-time*. The IPL offers a single, elementary communication abstraction: *unidirectional message channels*. Endpoints of communication are *send ports* and *receive ports*. The IPL then defines serialization and communication, and provides upper layers with interfaces to grid services such as topology discovery and monitoring. IBIS implements three application programming models on top of IPL: remote method invocation (RMI), group method invocation (GMI), and divide-and-conquer parallelism (Satin). Unlike many message-passing systems, the IPL has no concept of hosts or threads, but uses location-independent IBIS *identifiers* to identify IBIS nodes. A registry, called IBIS *Name Service*, is provided to locate peer networking endpoints, allowing to bootstrap connections.

As a consequence of the IBIS project premisses, that is, to provide application developers with an uniform programming and execution environment for grid computing, the network stack must be able to accommodate a wide range of heterogeneous scenarios, depending on the available network resources and connectivity discovered at run-time on each assigned platform. A communication system for grid computing therefore should have the following properties:

1. It should be dynamic, so it can run on networks that are not pre-determined at the time the application is launched; the protocol stack therefore should be *configurable at*

run-time;

2. It should be heterogeneity-enabled, so it can run on multiple different networks at the same time;
3. It should be efficient, so it can exploit HPC interconnects possibly found on the grid clusters on which the environment will run.

6.1.2 Design

NETIBIS is an implementation of the *Serialization and Communication* part of the IBIS IPL. To address the premisses and requirements above, the NETIBIS network stack is designed following a modular and composable architecture. The stack can be tailored according to both the networking resources discovered on the target platform, and the specific service requirements coming from the applicative code. This communication management layer is able to establish connections dynamically over hardware links ranging from high performance, local area networks —in particular, it was ported on MYRINET HPC networks at the time— to wide-area networks interconnecting clusters from several supercomputing sites [62].

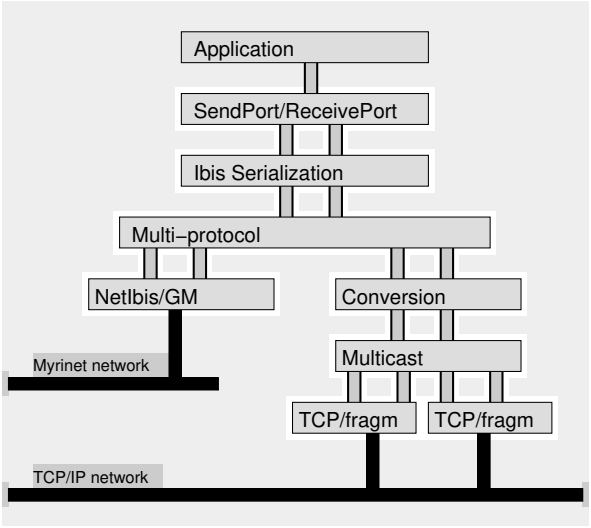


Figure 6.2 – The NETIBIS architecture.

NETIBIS supports protocol stacking using two fundamental concepts: connections and drivers. A *connection* links a `send` port on the source process, to a `receive` port on the destination process. They are established end-to-end from the IPL `send` port down through the NETIBIS modular protocol stack to the actual hardware link on the sending machine, and then up through the NETIBIS stack again on the receive side, to the IPL receive `port`. Internally, NETIBIS connections actually are made of two network links, however. The *application* link is a unidirectional link and uses the networking software/hardware selected by the user. It is exclusively used by the application. The *service* link is a bi-directional link made of a pair of streams [1]. It may be used by the NETIBIS internals to exchange data between the source and destination nodes of the connection, for connection setup information transfer and for

1. TCP streams in the reference NETIBIS implementation.

low-frequency synchronization. It is also used for dynamic negotiation (maximum transfer unit, buffer sizes), and it provides a rudimentary way for connection failure detection. A third category of network links, called *bootstrap links* [62], may be used in configurations where establishing a connection is particularly difficult, for instance when firewalls or NAT (Network Address Translation) are involved.

A protocol module in the communication stack is called a *driver*. *Filter* drivers implement network independent protocol functionalities. *Network* drivers implement support for specific low-level network APIs and hardware. Filter drivers are internal nodes in the protocol stack, whereas network drivers are leaf nodes. A driver consists of an *Output* object at the sender side or an *Input* object at the receiver side.

Available filter drivers include :

— **Serialization**: Transformation of JAVA objects into byte streams and vice versa; If the application only transmit JAVA arrays or elementary data types, this driver can be omitted from the stack.
— **Conversion (JAVA NIO)**: Optional use of JAVA's "*New Input/Output*" (NIO) I/O scheme.
— **Reliability enforcement**: Automatic packet delivery checking and retransmission driver, to be used over unreliable networks.
— **Fragmentation**: Cut communication stream into a sequence of data chunks of a maximum size.
— **Multiplexing**: Intersperse multiple communication flows into a single flow.
— **Multi-protocol**: Load-balancing of a communication flow onto several networking resources.
— **Multicast/multi-receive**: Message transmission to multiple receivers, message reception from multiple sources.

Figure 6.2 illustrates a NETIBIS protocol stack. It shows that a driver may act on multiple connections. Each NETIBIS `send` port and `receive` port may have its own stack configuration, but a connection may only be established between ports with matching stack configurations.

6.1.3 Discussion

The use of a composable communication stack for supporting distributed HPC applications may seem the source of an unnecessarily overhead, and the scenario of "Grid Computing" that was popular in the early 2000s looks a bit distant today —grids still exists, such as GRID'5000 in France and GÉANT in Europe, but are now rather oriented towards providing some federated and uniform access to geographically disseminated computing clusters, than running multi-cluster computing sessions—. Yet, several evolutions of the HPC landscape are opening new potentials for such kinds of stack. First, the emergence of the *Big Data* community around the development of techniques to processes huge amounts of informations, and second, the ongoing convergence efforts between such Big Data techniques and "legacy" intensive computing techniques, together bring issues that are similar to the NETIBIS context.

For instance, the APACHE SPARK environment proposes a programming model based on parallel data collections named RDD (Resilient Distributed Datasets). Parallel and distributed programs are expressed in this model by applying selection, computation and reduction operators on these datasets. SPARK is built using the SCALA language, which is itself running on top of the JAVA virtual machine. Distributed executions on top of this model lead to com-

munications involving expensive object serializations, and require flexibility in connectivity management to offer resilience. The communication stack could however be optimized, as for the communication stack of IBIS, to drive communications on top of a high performance network instead of a TCP/IP network when possible, and also to optimize or even bypass the serialization/de-serialization steps depending on the application needs. For example, an RDD array of Integer object items is very expensive to transfer because of the serialization requirements for the array items and the array itself. Representing this RDD as a native Integer array and leveraging the JAVA virtual memory's native I/O capabilities over a HPC interconnect would offer substantial performance improvements, as we observed with experiments on IBIS applications [19]. The requirements for SPARK to work on a large variety of very different hardware platforms is akin to the needs for the IBIS environment to work on diverse Grid Computing clusters. The capability of SPARK to add and remove computing nodes from running sessions and to implement resilience advocates an *open* session management model such as the one offered by NETIBIS. All-in-all, SPARK communication requirements present striking similarities with IBIS's requirements fifteen years before.

If the Grid Computing platforms are not too common anymore, "Cloud computing" platforms instead are very much widespread nowadays, and build on the legacy of grid principles. When running an applicative code on a rented computing cluster, as provided by Cloud companies, the applicative code may have to adapt to the reality of those resources obtained on each rented session. Some of that reality is often abstracted on cloud platforms, by using virtualization mechanisms. Yet, the connectivity, the communication devices virtually exposed within the virtual machines, and the observed performance characteristics may vary, leading to evolving communication stack needs within the applicative code from run to run. More generally, the heterogeneity inherent to platforms necessitating distinct kinds of nodes to interoperate may necessitate composable, customizable communication stacks, such as a main computing cluster connected with some visualization nodes, or with an in-situ front-end processing cluster to filter scientific tools output on the fly, or with some back-end storage nodes, or also with some intermediate non-volatile memory buffering nodes, for instance.

On the applicative code side, the evolution towards using multiple software modules together —such as a core application and specialized libraries— may lead to distinct network related requirements for each module. For instance, a distributed mathematical library such as SCALAPACK or its descendants may require a powerful HPC interconnect, while a deployment module or an interactive control module may find TCP/IP connectivity sufficient; a checkpointing module or a visualization module may have distinct needs in terms of priority, more general quality of service requirements, and reliability guarantee. A composable communication stack enables each module to get its own dedicated stack specially tailored to its purpose, while isolating the communication flow of performance critical modules from less demanding or low traffic modules.

6.2 Interoperable Runtime System Resource Management

In the case of the NETIBIS communication system, all the software modules constituting the individual building blocks of the stack are part of the same software project, and therefore naturally designed to work well together. A whole application however frequently needs to rely on multiple software elements designed independently. Unfortunately, runtime systems for HPC have long assumed having full, exclusive access to all hardware resources assigned

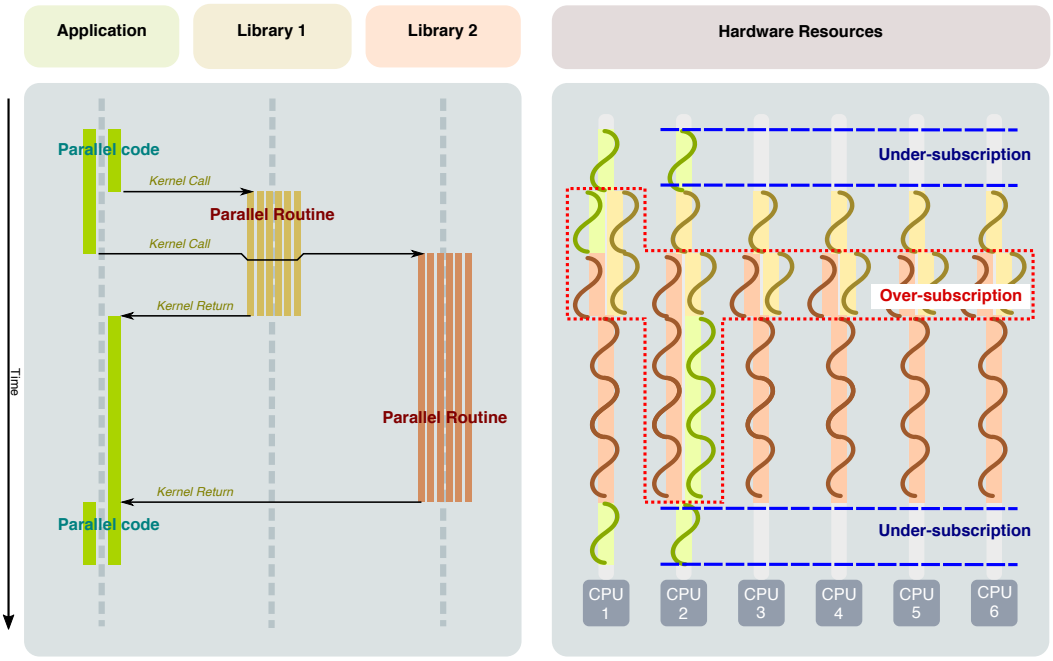


Figure 6.3 – CPU over-subscription on a multicore system, due to un-coordinated resource access between multiple parallel codes.

by the operating system. This assumption may lead to severe performance impacts, when multiple runtime systems are used together within the same application. One of the main purposes of the European H2020 INTERTWINE project [99] (2015–2018, *Programming Model INTERoperability ToWards Exascale*) was therefore to address this situation, by designing mechanisms through which multiple runtime systems conjointly used within the same code would be able to negotiate their access to hardware resources in an orderly, dynamic manner.

6.2.1 Premisses and Requirements

Due to the complexity of computing platform hardware and the diversity in HPC applicative code requirements, designing a single runtime system to address every possible needs is likely intractable, at least for the foreseeable future. Even within an application, it is increasingly common to employ more than one runtime system at the same time, either directly or indirectly. A typical example is an application utilizing a computing runtime system —for instance an OPENMP runtime— together with a communication runtime system —often an MPI implementation (see Section 4.1)—. Another common scenario is the case where the application relies on a computing runtime system for managing its own parallelism, and this application is also making calls to a library (for instance a library implementing linear algebra routines, such as a BLAS or LAPACK library [24, 7]) where routines are themselves parallelized using another runtime system (see Figure 6.3). In both scenarios, the two runtime systems involved do not coordinate their usage of available hardware resources: the threads

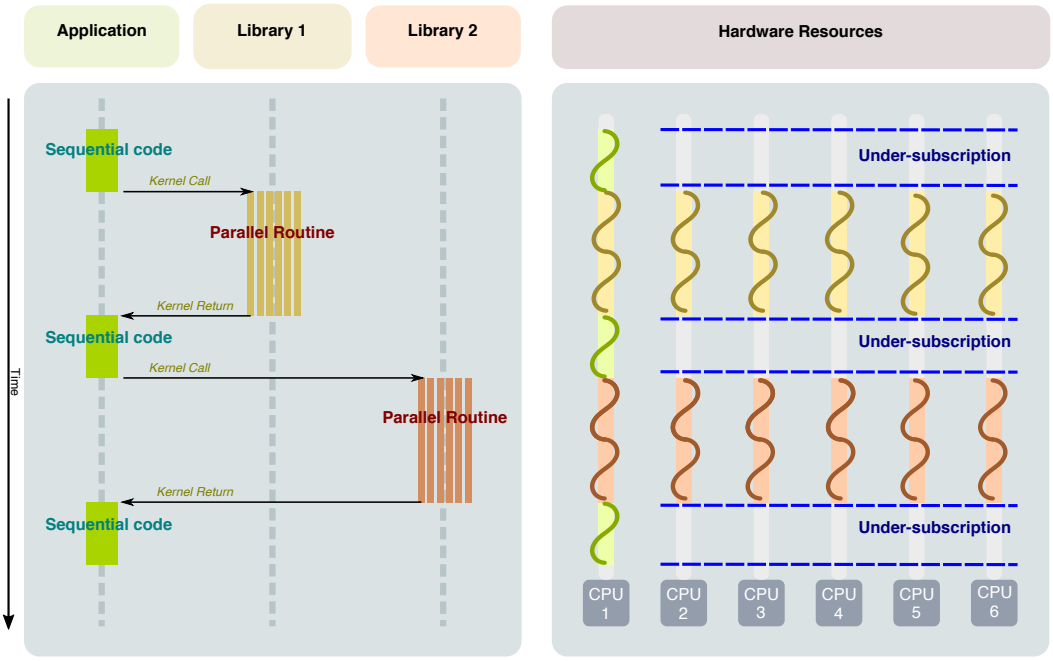


Figure 6.4 – CPU under-subscription on a multicore system, possibly due to an overly conservative application design.

launched by one of the runtime systems (for running computations, or for managing communication progression) would then conflict in accessing processor cores with the threads of the second runtime, resulting in over-subscription.

Avoiding such phenomenon usually requires the application programmer to perform some resource arbitration between these runtime systems, when this is possible, using mechanisms specific to each runtime system. This is harmful from a separation of concerns point of view and from a portability point of view. This may also results in performance degradation if the processor cores still remain oversubscribed, due to an erroneous setting from the programmer, or due to evolving resource needs throughout the lifespan of the application. On the other hand, being overly conservative, by exclusively using a sequential applicative code, and serializing calls to parallel kernels one at a time would also result in limited performance due to resource under-subscription (see Figure 6.4). Instead, one of the key objective of Project INTERTWINE, funded under the European H2020 program, has been to design an API [163] and define associated mechanisms, to enable multiple runtime systems to interoperate within the same applicative code by coordinating with each other in accessing hardware resources.

6.2.2 Design

The INTERTWINE resource management API [163] addresses several identified scenarios where a runtime system has to coordinate its access to hardware resources with another software entity, such as another runtime system, within the same applicative code. These

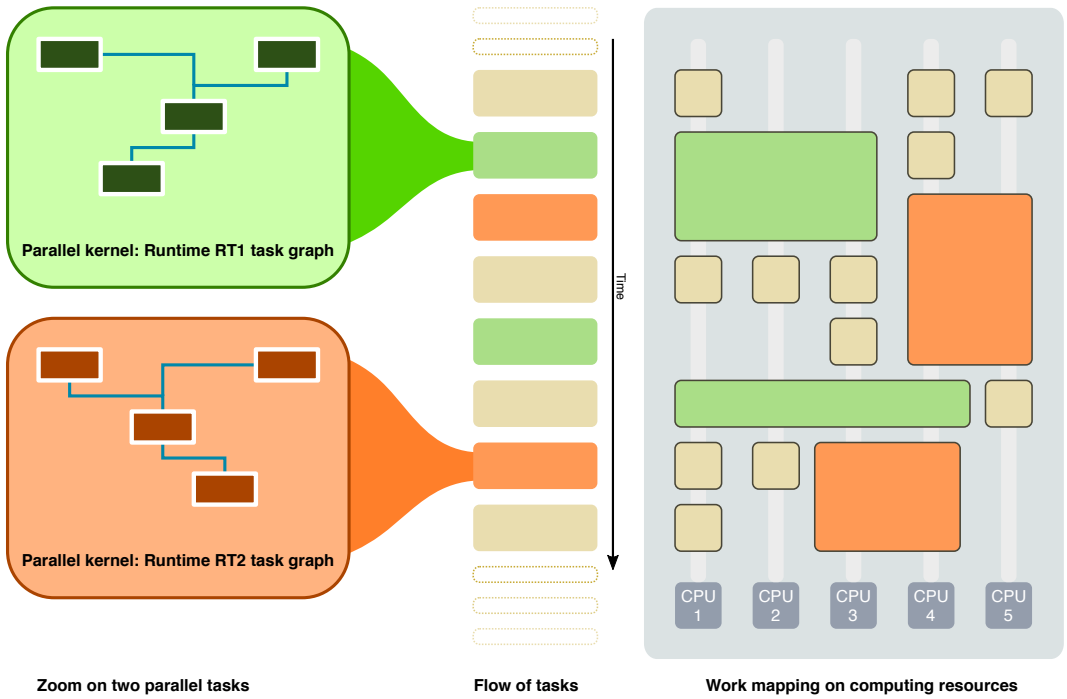


Figure 6.5 – Concurrent interoperability scenario between two parallel task-based runtime systems.

scenarios correspond to the nature of the relationship between the runtime system considered and the peer software entity with which it shares hardware resources:

Concurrent Relationship Two parallel computing runtime systems work alongside independently. This scenario occurs when the applicative code uses multiple parallel libraries or kernels parallelized with distinct runtime systems. It is illustrated on Figure 6.5. Figure 6.3 is an example of this scenario while resource usage is instead uncoordinated. To address this situation, the API defines a series of routines for runtime systems to negotiate access to computing resources. The API does not impose a limit on the number of concurrent runtimes; however, for the sake of the discussion, let's assume two runtime systems designated as `RT1` and `RT2` respectively.

The principle is the following. Each participating runtime system, `RT1` and `RT2` here, is first initialized with a subset of available hardware resources —possibly all resources, or even no resource at all—. Then, as the application execution progresses, if one runtime, say `RT1`, detects that one of its computing resources is about to become idle, it can notify other runtimes, `RT2` here, that the corresponding unit becomes available. `RT2` then has the option to borrow it if needed, for instance to bring in more computing power to go through a compute intensive phase. When `RT1` subsequently encounters a compute intensive phase itself, it can notify `RT2` about it, to reclaim any borrowed unit.

The two runtime systems can coordinate through the API either directly in a point to point manner, or using a third-party manager. The *dynamic load balancer* DLB framework [81]

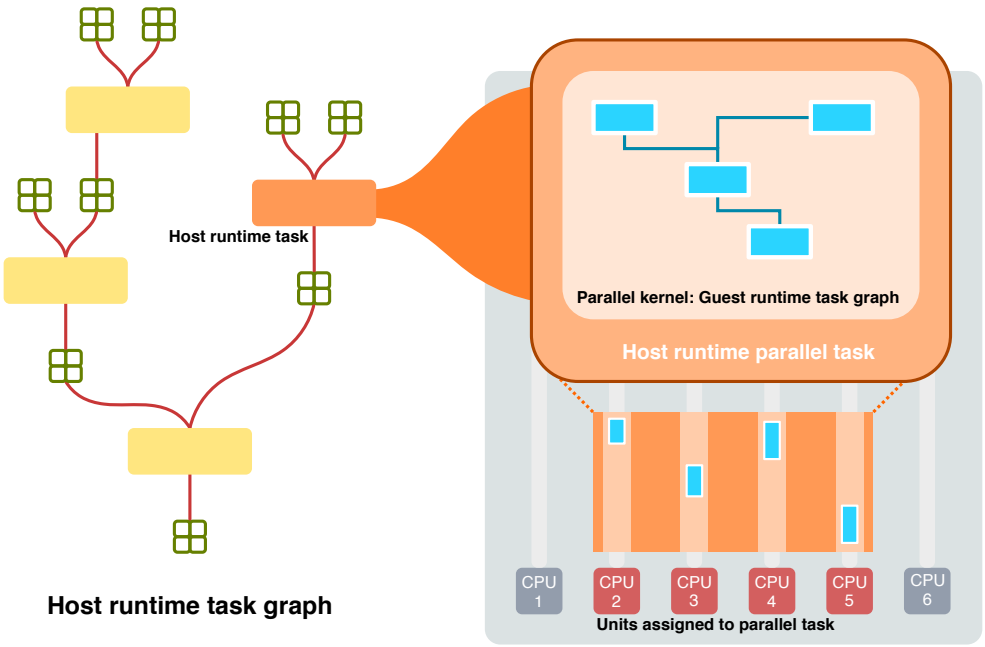


Figure 6.6 – Nested interoperability scenario between two parallel task-based runtime systems.

developed at the BARCELONA SUPERCOMPUTING CENTER, was used during the INTERTWINE project for that purpose. Using a third-party manager simplifies the development of each runtime, it plays a role of a "stock-exchange trade center" for computing units, that runtime system can target, in a transparent manner with respect to the number of participating runtime systems; this transparency is especially convenient for being used in a library.

Nesting Relationship Two parallel computing runtime systems work together following a host-guest scheme. This scenario, illustrated on Figure 6.6, occurs when a parallel routine from one runtime system is called within the context of a parallel construct of another runtime system, for instance when a parallel BLAS [24] routine is called from within a task in a task-based applicative code. In such a situation, the runtime system in charge of executing the parallel BLAS routine (that is, the *guest* runtime system) should do so over the hardware resources temporarily assigned by the *host* runtime system in charge of scheduling the tasks, so that the host runtime's accounting of resource usage and idleness is accurate.

Again considering two runtime systems `RT1` and `RT2` for the sake of the presentation, the principle of this part of the API is the following. If `RT1` is the runtime system of the caller code, and `RT2` is the runtime system of the callee code, the API defines some "offloading" methods, to launch the computation of some parallel computation managed by `RT2` on the computing resources selected by `RT1`. The idea is similar to an applicative code executed on the main CPU, offloading some computation on an OPENCL or NVIDIA CUDA accelerator board. Indeed, the API follows the analogy of the accelerator board by providing the programmer with an OPENCL-based offloading approach, beside the *native*, function call approach. The OPENCL-

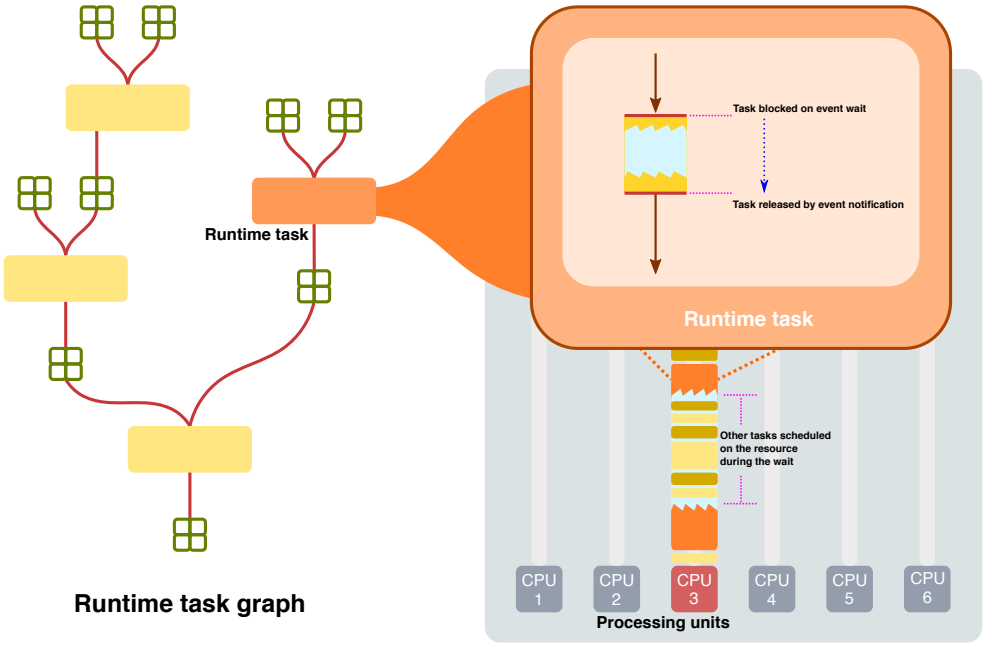


Figure 6.7 – Service relationship scenario between two parallel task-based runtime systems.

based approach presents the guest runtime system, RT2 here, as a pseudo-accelerator board for the host runtime system RT1.

Service Relationship The computing runtime system temporarily relinquishes ownership of some computing resources in wait for some external event, to be notified by some other runtime system or software entity. This scenario occurs when some task must block on some event such as an incoming communication message. While this scenario resembles a *Nesting relationship*, the issues are distinct: a task blocking on an event without relinquishing its assigned hardware resources is essentially wasting these resources, since the task cannot progress while the event is pending, but it prevents other works to be done on the resources it owns; a task calling a parallel kernel is not in the same situation since its assigned resources can and should be used by the nested parallel kernel.

From an implementation point of view, the nesting relationship scenario could (but not necessarily would have to) be implemented by calling the nested parallel kernel on the software stack(s) of the participating worker threads of the host runtime system. The service relationship, instead, would:

— Either involve temporarily de-scheduling the worker threads of the runtime system during the time of the wait, in order to release the associated computing resource for it to be used by another worker thread;

— Or use some deferred execution mechanism such as continuation passing, or future/promise, for instance, where the asynchronous event would typically instantiate the *continuation*, or resolve the *future* through the *promise*.

The most common usage for the service relationship is a computing runtime system making use of some communication framework. However, it is also, naturally, convenient to interface with some event-loop based framework. Moreover, it enables some rudimentary "one-way" resource interoperability with external software not interfaced with the INTERTWINE resource management API.

6.2.3 Discussion

The task of assigning computing resources to applicative codes traditionally belongs to the operating system scheduler. However, for the specific context of high performance computing applications, the OS scheduler is not sufficient to perform this role. When an user launches a HPC application on a supercomputer, that application typically obtains an "almost" exclusive usage of a node's resources for a duration that can reach hours, days, or even months. From, the point-of-view of the operating system on one node, a single process or set of processes belonging to the same unique user are to be scheduled, without any information on their workload or characteristics. If the process' threads or processes' threads are bound to CPU cores, as is typically done nowadays, the task of the OS scheduler is reduced to its strict minimum, except perhaps for executing some system daemon from time to time. Therefore, as discussed in Section 3, HPC applications often resort to using computing runtime systems to complement the OS scheduler, and one could say to virtually supersede it for most practical purposes, in managing the fine grained applicative code work mapping on computing resources.

Yet, for modern, composite HPC applications, even a computing runtime system may get an incomplete view of resources needs and usage. Multiple applicative code components may each rely on distinct runtime systems, or try to initialize multiple instances of the same runtime system. Some specific components such as communication libraries may also bypass any runtime system and directly allocate threads on their own, as noted above. When these interference situations occur, they usually result in performance degradation due to resource over-subscription (such as two runtime systems assigning work to the same CPU core) or under-subscription (a CPU core kept idle by a runtime system while another runtime would have work to assign it). Hence, there is an increasingly pressing need for multiple runtime systems and libraries to actively cooperate in accessing resources. The API elements proposed above constitute some possible building blocks to implement such a cooperation. However, a joint effort is likely needed within the HPC software ecosystem as a whole to take the resource-related interoperability issue into account.

While we used the example of CPU core assignment, the interoperability effort should encompass a broader view of computing resource sharing. The INTERTWINE resource management APIs indeed support negotiating accelerator devices as well. Yet, we could go further, by integrating negotiated resource management for memory, and especially for specific memory spaces —such as some high bandwidth memory/HBM area, for instance— that may be highly demanded by multiple runtimes while only available in limited amount. The memory of accelerator devices could also be managed in a shared manner, to avoid flushing all the on-device data upon reassigning a device from one runtime system to another one.

Once runtime systems are able to non-exclusively and cooperatively manage resources, however, there still needs a way to guide and control the cooperation. The Process Management Interface – Exascale (PMIX) [48] initiative, initially originating from the MPICH

project, is an interface and a framework for deploying applications processes and establishing connectivity between them on supercomputing clusters. It typically interacts with the job scheduler of a cluster to drive the launch of an application job on the nodes assigned by that job scheduler. Such a framework could likely constitute a key actor in controlling resource-interoperable runtime systems within the application processes, by assigning an initial resource distribution to each runtime within a node, and enforcing resource sharing policies throughout the execution.

On the opposite side, a runtime system should also accept to serve multiple applicative code parts. StarPU, for instance, defines the notion of *scheduling context* [97, 55]. With this notion, each distinct applicative part can get its own scheduling context and entirely ignore that other applicative parts are concurrently using other scheduling contexts from the same StarPU instance. The StarPU engine then manages resources between all these contexts. This concept of context is becoming more widespread [66]. However, work still remains also, for example with the MPI and the OpenMP standards. MPI defines the concept of *communicator* that can be used in a way close to contexts by assigning distinct communicators to distinct applicative parts. Yet, each applicative part commonly hard-wires the use of the `MPI_COMM_WORLD` main communicator, and there is no transparent mechanism to assign distinct communicators to distinct applicative parts to isolate them from each other. Some alternatives are being considered by the MPI Forum [65, 86], but have not yet been adopted, at the time of this writing.

The OpenMP specification does not yet define a dedicated support for letting multiple applicative parts access the OpenMP runtime system concurrently, in an isolated and controlled manner. Thus, there is no specific provision for supporting an OpenMP application that would call a library routine also parallelized with OpenMP: The resulting process might work well or show performance issues due to interferences, or possibly worse if the applicative code and the library are not compiled with the same compiler and do not use the same OpenMP runtime system. In contrast, the specification defines some support for an OpenMP runtime system to temporarily relinquish some or all the resources it owns upon request of the applicative code through the `omp_pause_resource()` and `omp_pause_resource_all()` standard routines. The language also defines a `pragma omp interop` construct. However, at the time of OpenMP 5.1, this construct is currently limited to interoperability with so called "foreign" tasks in foreign execution contexts on accelerator devices, such as a cuBLAS routine, for example. Possible evolutions of this mechanism might also include establishing task dependences with external task-based runtime systems, or even letting OpenMP tasks wait for external events such as receiving a message; some work has for instance been engaged by the OmpSs team at BSC with the TAMPI (task-aware MPI) mechanism [142].

6.3 Conclusion on Interoperability

In this chapter, we studied two pieces of work —one conducted during my post-doc in the team of Pr Henri Bal at Vrije Universiteit, Amsterdam, The Netherlands, on the Ibis project, and another one conducted as part of the European project H2020 INTERTWinE, for which I represented Inria— involving the notions of interoperability and composition in a HPC context, as means to enhance programmability. Indeed, it is usually more convenient and considered better practice to design small pieces of software each focused on a specific purpose, than write monolithic code. This, however, entirely relies on the possibility to assemble such

individual components into a composite applicative code. Each component must therefore be carefully designed to assemble with other components by following some dedicated API. Moreover, the resulting composite code must behave efficiently, and interferences must be avoided between any pair of components by coordinating their access to resources. The first piece of work we explored was the NETIBIS composite communication stack for the IBIS project, and the second piece work studied was the design of INTERTWINE's cooperation programming interfaces between multiple runtime systems —here STARPU and OMPSS— to let them dynamically adjust their computing resource allocation according to the evolving needs of the applicative code, throughout the application lifespan. We then discussed the current status of the interoperability and composition concepts within HPC, and especially the need to make a larger number of runtime systems composable and interoperable.

Next chapter will now study the enhancement of HPC code programmability with a programming environment attempting to separate hardware-specific optimizations and memory management operations from the natural, architecture-neutral application algorithm.

Chapter 7

Enhancing Programmability through Separation of Concerns

In this chapter, we present the INKS programming model [72], designed and developed by Ksander Ejjaaouani during his PhD Thesis (2015–2019) [71], of which I have been one of the advisors. The INKS model is based on the principle of an *algorithm* description language working in cooperation with *optimization* description languages, in order to enforce the separation of concerns. With such a model, a canonical algorithm can be expressed independently of the instantiation and optimization choices, while such choices are kept "physically" separated in a side file of instantiation directives. This model thus shares some of the goals of the STARPU task-based runtime system, where the flow of tasks is also expressed independently of the execution choices of these tasks. It does so with a different approach and a different scope, though: INKS mainly targets iterative simulations, expressed as an enclosing time loop, and some loop nests accessing some arrays in a compliant manner with the polyhedron model [77, 76]. STARPU instead targets more irregular algorithms.

7.1 Context

The nature of current high performance processing technologies (see Section 3.1) necessitates suitably tailored applicative codes to deliver all their potential benefits. This is especially true to exploit hardware parallelism capabilities and cache memory hierarchies, that both have a critical impact on performance. This requirement for specially tailored applicative codes unfortunately tends to force programmers to deal with and master technical matters that bear little to no relevance for the algorithms they are implementing. It is indeed unfortunate that a scientific programmer designing a computational fluid dynamics code or a molecular dynamics code should have to take into account the potential for vectorization or the locality of memory accesses in a numerical kernel loop nest. Moreover, this interleaving of domain specific algorithmic concerns and computer-related technical concerns leads to an application code difficult to read, to maintain over time, and to port on new architectures.

Many approaches have been proposed to improve this situation, some of them presented in chapters of the present manuscript, mainly through the principle of delegation, to a library —for instance a runtime system, a communication library, a specialized numerical library such as LAPACK [7] or FFTW [79]— or to a compiler applying some predefined optimization strategies on the source code. The compiler can be a general purpose language compiler

(e.g. a C, C++ or Fortran compiler, for instance) or a dedicated domain-specific language compiler [160, 53, 60].

Other, more specific approaches exist also. Skeleton-based approaches such as SKEPU [74, 73], SKETO [161] or LIFT [153], offer some pre-defined support for common kinds of parallel algorithms, following the concept of *algorithmic patterns* used in the domain of software engineering. Skeletons drastically reduce the amount of redundant programming work, and directly contribute to separation of concerns enforcement, provided though, that the intended algorithm cleanly falls within the families of supported skeletons; having to write a new skeleton from scratch may instead cancel some of the benefits of such a framework. Domain Specific Languages (DSLs) are even more dedicated approaches, that can for instance target a single specific family of algorithms such as POCHOIR [160] or PATUS [53], or even a specific kind of application such as the QIRAL [22] language for Quantum Chromodynamics (QCD) simulation.

7.2 The INKS Programming Model

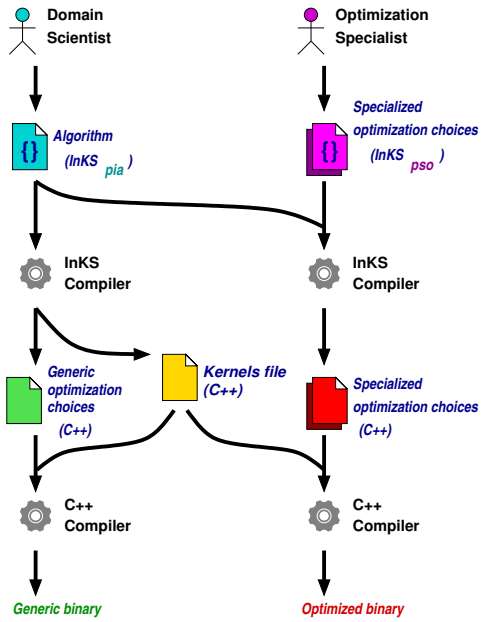


Figure 7.1 – Principles of the INKS model.

The INKS model we designed [72] uses a distinct approach. This approach is inspired by aspect-oriented programming (AOP) languages such as LARA [42] (for embedded systems), whose purpose is to enforce a strict separation of multiple orthogonal concerns while writing applicative code. The principle of INKS is to separate the expression of the "*raw*" algorithm on the one side, from optimization-related technical choices on the other side. This separation is materialized by the use of two distinct pieces of code, one expressing the raw algorithm and one expressing optimization choices. These two pieces of source code are then integrated together using a dedicated compiler.

Within the INKS model, an applicative code is an algorithm simulating (for instance) some physical phenomenon, which consists in: the simulation inputs; the set of values computed; the formula used to produce each of these values; the simulation outputs. The *optimization choices* are the technical choices that may affect the performance of the computation without altering its result, which depends on the *simulation algorithm* only. This may be for instance the choice of the computing unit to use for a given piece of computation, the choice of a memory location where to put some data accessed during the computation, or the choice of the ordering of some computations with respect to other computations.

The INKS model is depicted in Figure 7.1. The INKS_{pia} language (PIA for *platform-independent algorithm*) enables the application programmer to express some simulation algorithm independently of optimization choice concerns. Then a family of INKS_{pso} optimization description languages (PSO for *platform-specific optimizations*) is available for an optimization specialist to specify efficiency-related instantiation choices, in a distinct source file. Both source files are then processed by the INKS compiler, and the resulting output sent to any C++ native compiler available, to produce an optimized executable. Several optimization choice alternatives can thus easily be explored for a single algorithm, for instance to optimize for distinct target platforms. A vanilla, INKS_{pia}-only version can also be generated for validation and debugging purpose.

Listing 1 – 1D stencil computation on a 2D domain in INKS_{pia}.

```
1  kernel stencil3(x, t):
2  (
3     double A(2) {in: (x-1:x+1, t-1) | out: (x, t)}
4  )
5  #CODE(C)
6     A(x, t+1) = 0.5 *A(x, t-1)
7               + 0.25 * (A(x-1, t-1) + A(x+1, t-1));
8  #END
9
10 kernel boundary(x, t):
11 (
12    double A(2) {in: (x, t-1) | out (x, t)}
13 )
14 #CODE(C)
15    A(x,t) = A(x, t-1);
16 #END
17
18 public kernel inks_stencil(X, T):
19 (
20    double Array(2) {in: (0:X, 0) | out: (0:X, T-1)}
21 )
22 #CODE(INKS)
23    stencil3 (1:X-1, 1:T) : (Array),
24    boundary {(0, 1:T), (X-1, 1:T)} : (Array)
25 #END
```

The INKS_{pia} Language Listing 1 illustrates the INKS_{pia} language on an example. INKS_{pia} is based on the dynamic single assignment (DSA) abstract model: computed values are stored in conceptually "infinite" multidimensional arrays, where each coordinate can only be written to at most once. The mapping of such abstract arrays on real memory locations is left unspecified in INKS_{pia} due to the separation of concerns, since such a choice deals with instantiation and optimization.

Computations are expressed as *kernel* procedures. Such procedures expect some arrays and some integer coordinates as input parameters, and declare some *validity domains* specifying the coordinates they might read from and/or write results too. They also implement a piece of algorithm, either in C/C++ language or in INKS. Listing 1 shows a C language kernel `stencil3` and an INKS kernel `inks_stencil`. The execution ordering of a kernel is left unspecified within its validity domains. The `public` keyword identifies a kernel service as the simulation entry point (`inks_kernel` in the example).

As a whole, an INKS_{pia} listing specifies a parameterized task graph (PTG) [56], as for instance the PARSEC runtime system [35]. The PTG representation has the advantage to offer a compact, abstract representation for a large family of problems. In return, it also implies some constraints: in particular, all the problem parameters must be known at the kernel launch time, thus it is not possible to express adaptive meshes or adaptive time steps within the model. However, it is always possible to express them outside and call INKS kernels multiple times afterward.

Listing 2 – 1D stencil computation on a 2D domain: instantiation directives in INKS_{XMP}. and corresponding generated code from Listing 1 INKS_{pia} + this INKS_{XMP} code

```
/* InKSo/XMP specification */
#pragma xmp nodes p[nNodes]
void inks_stencil(T& Input, size_t X, size_t T){
      #pragma inks decompose % Array // allocation of "Array" algorithmic array
            (2*2, 1) // dimension reordering, time dimension is fold by 2 and not distributed
            with tmpl onto p // block decomposition mapped on the XMP topology

      for(int t=1; t<T; t++){
            boundary(Array, 0, t);
            for(int x=1; x<X-1; x++)
                  stencil3(Array, x, t);
            boundary(Array, X-1, t);
      }
}

/* - - - - - */

/* XMP + C result */
#pragma xmp nodes p[nNodes]
#pragma xmp template tmpl[:][:]
#pragma xmp distribute tmpl[*][block] onto p
void inks_stencil(T& Input, size_t X, size_t T){
      #pragma xmp align Array[t][x] with tmpl[t][x]
      #pragma xmp template_fix tmpl[2][X]
      Array = (double(*)[X]) xmp_malloc(xmp_desc_of(Array), 2, X);
      for(int t=1; t<T; t++){
            boundary(Array, 0, t);
            for(int x=1; x<X-1; x++)
                  stencil3(Array, x, t);
            boundary(Array, X-1, t);
      }
}
```

The INKS Optimization Language Family While the INKS compiler can generate a valid program from the INKS_{pia} source, this program will be unoptimized. To apply optimization directives onto the vanilla INKS_{pia} source code, we explored several complementary domain specific languages collectively referred to as INKS_{pso} languages. As a proof-of-concept, the

INKS$_{loop}$ language offers some directives to specify loop nest ordering and blocking parameters. From a unique, unmodified INKS$_{pia}$ kernel, it enables experimenting with multiple strategies, and selecting distinct preferred orderings for different underlying hardware platforms. It can also insert vectorisation-related directives to be used by the back-end native compiler.

The INKS$_{XMP}$ domain specific language is a more advanced INKS complement language, which enables an INKS kernel to take advantage of the XcalableMP [119] directive-based language (often abbreviated as XMP). XMP is developed in Japan. Revisiting some concepts of High Performance Fortran (HPF [135]) and Coarray Fortran (CAF [123]), such as data layout and data distribution expressions (inspired from HPF), array expression extensions (inspired from CAF), all in a consistent manner, it offers a parallel and distributed programming environment using *pragma* directives on top of the C language (and also on top of Fortran, though we do not use this capability with INKS). It handles data structures such as distributed arrays and it manages data transfers transparently by targeting a communication system such as MPI. Thanks to XMP, the INKS$_{XMP}$ language enables parallelizing and distributing the computation of an INKS kernel in a non-intrusive manner. Listing 2 shows an example of an INKS$_{XMP}$ companion source file to the INKS$_{pia}$ Listing 1. The INKS$_{XMP}$ listing basically parallelizes each time iteration of the stencil computation on all participating nodes, using a double buffering scheme obtained by folding the time dimension of the abstract array.

Building on these case-specific prototypes, Ksander Ejjaaouani designed a more generic optimization definition language, simply named INKS$_{pso}$, to conclude his thesis [71]. This language, defined in Chapter 5.4 of Ksander's thesis, enables specialist developers to design optimization strategies to be applied on INKS$_{pia}$ codes. For that, it offers directives to specify memory layout mappings on the one side, and to control computation orderings on the other side: that is, the instantiation decisions abstracted-away in INKS$_{pia}$ kernels.

7.3 Discussion

7.3.1 Broader Scope through Composition

With the INKS$_{pia}$ and INKS$_{pso}$ language, the INKS environment reached a proof-of-concept milestone on implementing separation of concerns between the generic process of algorithmic development and the often technical HPC-oriented optimization process. The environment is however exclusively oriented on a static approach so far, due to the requirement that the polyhedron model applies on static control parts. Therefore, an important next step would be to integrate some aspects of dynamic adaptiveness within the approach, while taking care of preserving the benefits brought by the polyhedron model. The PARSEC runtime system, for instance, when used through its PTG programming model, combines a compiler-based approach to generate the compact, parameterized graph of tasks with a run-time work-stealing scheduler at the node level, in order to drive the execution on multiple cores in a balanced manner within each node. Thus, an INKS statically optimized coarse scheduling could be refined at run-time by a dynamic scheduler to smooth unforeseen bumps in the execution flow.

The current INKS model has been experimented on simulation codes with a time loop and nested kernels. The use of composition techniques discussed in Section 6, and mainly in Section 6.2 could enable supporting a wider range of applicative codes. Through the

nesting composition relationship, an INKS kernel could leverage routines expressed using techniques outside of the INKS scope, provided that such routines do not exhibit hidden dependences, in the same way that for instance the high level PYTHON language can leverage highly optimized routines from NUMPY even though implementing such routines is outside the scope of PYTHON. Conversely, an INKS applicative code can naturally be embedded in a larger non-INKS applicative code as well. Through concurrent composition relationship, and beyond the possibility to run some INKS applicative code side-by-side with another code on distinct computing resources, the model would have to be extended to support some kind of kernel heterogeneity —that is, support applying one kernel on some parts of the domain and another kernel on some others— and more generally to support conditionals, which it does not support at this time.

Composition also lets alternate applicative phases with distinct properties in terms of static vs dynamic parameterizing. We discussed in Section 3.4.1 how distinct task programming models enable discovering tasks at compile time through a PTG representation, or on the fly at run-time. The INKS model itself is based on PTG as explained above. It would therefore benefit from cooperating with a dynamic discovery-based environment, such as provided by the Sequential Task Flow (STF) programming model of the StarPU runtime system for instance, to support classes of applicative codes featuring alternated statically parameterized vs dynamically parameterized phases. In return, informations obtained through INKS about the statically parameterized parts could benefit the runtime system in charge of the dynamically discovered parts, especially in optimizing phase transitions, for instance by triggering some data prefetch operations.

7.3.2 Data Layout and Kernel Optimization

Besides the aspects related to blocking and data reuse, the matter of data layout has not yet been studied in the context of INKS and constitutes another important future work. Indeed, as exposed in Chapter 9, the choice of an appropriate data layout has an impact on performance, and is interdependent with optimization choices. For instance, a data layout oriented towards structures of arrays (SoA) is usually more efficient than a layout oriented to arrays of structures (AoS) when using vectorization/SIMDization code optimizations. Thus, the INKS optimization back-ends could be extended with multiple data layout strategies, and exploit the benefits of the separation of concerns principle to let the programmer experiment with such different data layouts.

Moreover, INKS provides programmers with a natural input to the BOAST [170] kernel optimization framework. BOAST, developed by Brice VIDEAU is a framework designed to generate series of kernel instances with parametrized, varying optimization strategies, for autotuning purposes. It also has the possibility to generate accelerator codes for CUDA and OpenCL GPUs. It is thus specialized on the complementary part to the INKS_{pia} architecture independent algorithm expression language. As such, the combination of them looks promising.

7.3.3 Heterogeneous, Distributed Platforms

Exploiting distributed platforms is an essential capability, and the INKS_{XMP} prototype built on top of the XCALABLEMP language demonstrates the ability of INKS to handle distributed computing without altering the separation of concerns and the properties of the

model. The XCALABLEMP language is especially a good match as a foundation for an INKS optimization backend, because of its pervasive design philosophy of explicit operations. The XCALABLEMP associated runtime system does not take any action on its own, and every operation has to be duly requested with a corresponding directive, a result of the lessons learned from the HPF era. When working with INKS, this gives the INKS XCALABLEMP backend a full control over operations, without risks of interferences. On the INKS side, XCALABLEMP brings the advantage that common communication schemes involving array data structures are abstracted by the XCALABLEMP language, such as the handling of shadow cells exchange on partitioned array borders, that are ubiquitous in the classes of simulation applications targeted by INKS. Moreover, since XCALABLEMP is itself compiled to MPI routines, the overhead introduced by the XCALABLEMP layer is small. Overall, this is a particularly compelling example of the interest of separation of concerns.

Extending the INKS model to support heterogeneous, accelerated architectures stands among the natural follow-ups of this work. Indeed, INKS could help in the process of generating codes for distinct computing units, such as a CPU and a GPU, by letting apply separate optimizations choices through INKS_{pso} to a unique INKS_{pia} algorithm. The use of a framework such as BOAST cited above would make it relatively straightforward to integrate this capability. Extending INKS to actually drive the execution of some applicative code on a heterogeneous platform would require more work on the scheduling side, to take into account the relative efficiency of each computing unit as well as the data transfer costs, that are not currently considered in INKS' scheduling. Due to the complexity of generating such a heterogeneous scheduling, resorting to a hybrid static/dynamic approach and composing with a runtime system such as STARPU would make the task much more tractable.

7.4 Conclusion on Separation of Concerns

In this chapter, we studied the INKS programming model —conceived as part of the PhD work of Ksander Ejjaaouani— designed to let the process of writing a fundamental, platform-independent application algorithm, on the one side, and the process of making platform-specific optimization decisions, on the other side, be two separate concerns. This model aims at enabling programmers with different skills in terms of application domain development and in terms of optimization techniques for HPC platforms to cooperate on the development of a scientific simulation application. The benefit is twofold, in that it enhances programmability by letting more people engage in HPC application development, and it enhances portability in that it avoids platform specific developments to get irremediably entangled in the application algorithm. We then discussed possible extensions to the model, to accept a wider range of applications, to increase the scope of supported optimizations, and to target more complex hardware platforms.

This chapter also concludes the second part of this manuscript, in which we have examined several ways to enhance the programmability of HPC applicative codes, first through the abstraction of a parallel language such as OPENMP, then through the composition of parallelized codes into composite applications, and finally through the separation of programming and optimization concerns provided by the INKS model. The third and last part of this manuscript will now present several works about tuning strategies for bringing different classes of apparently ill-suited applicative codes back within the scope of high performance

computing.

Part III

Approaches to Performance Tuning

Table of Contents

Performance portability (see Part I) aims at reducing the amount of necessary programming efforts and optimizing efforts from HPC applicative code developers. Still, the development task on HPC platforms is sufficiently complex to necessitate dedicated help on the programming effort by itself (see Part II), and on the code optimization effort as will be explored in this third and last part.

Task-based runtime systems handle some part of the optimization job by applying scheduling algorithms to execute a series of tasks concurrently on multiple processing units, while enforcing constraints to ensure correctness. Yet, the issue of deciding on the granularity of tasks, that is, the computational weight in terms of number of instructions of each task, remains under the responsibility of the applicative code programmer. The granularity choice influences performances: a large number of lightweight tasks may overwhelm the task scheduler and cause a low ratio of task computation time over task management time; a small number of heavyweight tasks may instead limit the action of the scheduler and cause load imbalance. Chapter 8 presents a work we conducted on designing aggregation rules to build larger tasks out of many small tasks [138].

Compilers certainly account for the largest share of go-to optimizing tools, with ever increasing optimization capabilities. Assessing the effectiveness of a given compiler on a given applicative code, however, is a daunting task, which necessitates the exploration of the assembly code generated by the compiler. Chapter 9 presents some work we carried out on improving the task of analyzing compiler-generated assembly code using the MAQAO [23] binary analysis and patching framework, and hint at some potential modifications to be implemented on the applicative source code side to improve the resulting assembly code generated by the compiler.

In some situations, the code execution path can dramatically be simplified using case-specific knowledge at compile time. This strategy, named code specialization, can be employed to improve performances when a given use-case is used repeatedly, to avoid recomputing conditionals that are known to be constant for that case, to select faster paths or implementations taking advantage of the case peculiarity, to apply a wide range of common compiler optimizations (loop unrolling, dead code elimination, etc.) unlocked thanks to additional constants knowledge (e.g. perform general constant propagation techniques). We successfully applied the code specialization to the context of Error Correction Code (ECC) simulation to speed-up the validation of ECC algorithms [47], where a set of algorithm parameters specifies an ECC code instance which is then repeatedly applied on a large number of message frames. Chapter 10 presents this work.

Chapter 8

Tuning through Task Granularity

In this chapter, we explore the issue, identified in Section 3.5.2, of performance tuning through the selection of the "right" computational weight of tasks in task-based parallel applications, that is, the granularity that minimizes the task scheduling overhead costs, while maximizing load balancing capabilities. This study, presented in more details in [138], is conducted for the situation where the natural task parallelization of an application generates tasks much too light to be worthwhile to be scheduled individually. It investigates the definition of operators to aggregate tasks into heavier, more cost-effective composites.

8.1 Context

Within the context of the PhD Thesis of Corentin Rossignon [137] at TOTAL SA Company, which I co-supervised, we conducted a study on the adaptation of the granularity of tasks, in a task parallel iterative solver. This solver targets large sparse linear systems of equations preconditioned by an incomplete factorization [139], used in an oil reservoir simulation application. The preconditioner, involving an ILU incomplete factorization step and a triangular solve step of a sparse matrix, is representative of the difficulty of fine-grain parallelization: the fine grain description of these two steps' algorithms is the natural way to express the problem for the application programmer; however, in practice, a straight task-based parallelization of this fine-grain algorithm expression does not deliver good speed-ups. Indeed, because of the very low computational weight of the resulting tasks, the management cost of creating and scheduling individual tasks becomes prohibitive.

8.2 Task-Grain Tuning Through Guided Aggregation

We therefore designed an intermediate layer named TAGGRE [138] between the applicative code and the task parallel programming environment to offer means for tuning the computational weight of tasks in an accompanied manner. It is written in C++ and targets the INTEL threading building blocks (TBB [114]) task-based computing runtime system. The key idea of TAGGRE is to let the application programmer pack multiple individual algorithmic pieces of work into heavier tasks, by systematically applying sequences of generic, pre-defined task aggregation patterns, before they get submitted to the task scheduler.

TAGGRE expects a fine grain task DAG as input. From this fine-grained DAG, TAGGRE

builds a coarse-grained DAG using *aggregate operators* on subsets of tasks. The default aggregate operators implementation merely serializes the task functions called within all merged tasks. However, this behavior can be modified by overloading the aggregate method of TAGGRE's task class, for instance, to apply techniques such as loop unrolling.

8.2.1 Aggregate Operators

As a proof of concept, we defined several aggregate operators, each implementing one possible strategy to coarsen the DAG nodes. Multiple operators may be chained together to coarsen the DAG further. Each operator is designated by a letter. The sequence of selected letters is called the *Coarsening String*. For example, the Coarsening String "`SD(300)F(32)`" first applies the *Sequential* operator `S`, followed by the *Depth front* operator `S` with parameter 300, and finally by the *Front* operator `F` with parameter 32. The parameters are operator-dependent.

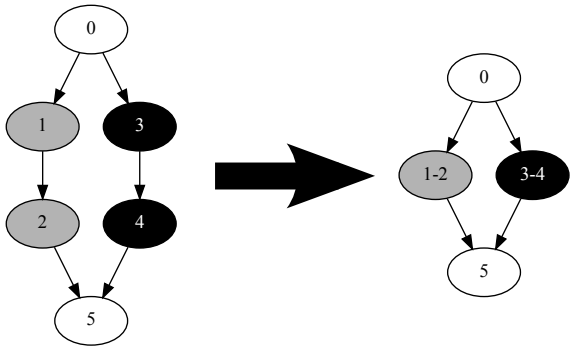


Figure 8.1 – Example of aggregation with operator `S`.

Sequential (S) The *Sequential* operator `S` is the most basic operator. It aggregates each task having a single predecessor with that predecessor, when such a predecessor has a single successor (Fig. 8.1).

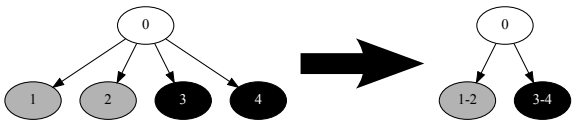


Figure 8.2 – Example of aggregation with the operator `F` and parameter 2.

Front (F) The *Front* operator `F` expects one argument N specifying the maximum number of tasks per depth level. The idea of this operator is to reduce the number of simultaneously available tasks. To do this, the Front operator implements a breadth-first traversal of the DAG. During the traversal, it aggregates tasks of same depth to build up to the maximum of N coarse tasks of similar computation time (e.g., similar grain size) per depth (Fig. 8.2).

Depth Front (D) The *Depth front* operator `D` expects one argument M specifying the maximum number of fine grain tasks aggregated into a coarse grain task. The main idea of this

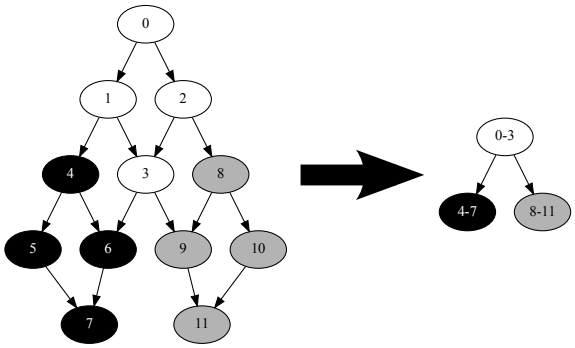


Figure 8.3 – Example of aggregation with the operator `D` and parameter 4.

operator is to aggregate a task with some of its descendants, up to the limit M. For that, the operator performs a breadth-first traversal of the descendants of a task to aggregate up to M tasks together. However, during the traversal, each new level encountered is sorted, from the task having the highest number of predecessors in the current aggregate being built, to the task having the least predecessors in this aggregate. The rationale of this operator is to favor aggregating tasks that are more tightly connected in the DAG (Fig. 8.3).

Continuation Oriented (C) The Continuation Oriented algorithm is an aggregation method which improves temporal data locality affinity by aggregating tasks working on data residing on close memory locations, in identical or close topological memory level (e.g. same NUMA bank, for instance).

8.3 Illustration of the Impact on Performance

We present here some evaluation results we conducted at the time, to illustrate the influence of granularity on task parallelism. However, since both hardware and software used at that time are now outdated, the raw performance measurements should not directly be compared with the nominal performance of nowadays equipments. The testing machine used for these evaluation is equipped with two Intel Xeon X5570 (Nehalem) 4-core sockets with 24 GB RAM (12 GB per socket) running Red Hat Enterprise Linux 5.2 with a Linux kernel 2.6.18. The compiler used is Intel C++ compiler XE 13 with level 3 optimizations. The back-end task scheduling runtime is Intel TBB. We also obtained comparable results using OpenMP tasks.

We perform the tests on two linear systems. The first one, Cube_100, is a system obtained from a 7-point finite volume discretization scheme of regular 3D cubes, with 100 points along each dimension. It is a scalar system: each row of the matrix is a vector of coefficients stored in double precision floating point. The second one, SPE10, corresponds to a 7-point discretization of a reference problem from the petroleum industry [150]. In this case, each entry of the matrix is a small dense $(3, 3)$ block of coefficients stored in double precision. Due to the geometry of the problems, the task dependence graph for Cube_100 is very regular, whereas it is an irregular one in the SPE10 case. The characteristics of the matrices are listed in Table 8.1.

Each fine grain task performs one elementary line operation. However, SPE10 tasks are

Table 8.1 – Matrix characteristics for each test.

Name	**Cube_100**	**SPE10**
Rows (n)	1,000,000	3,283,263
Number of non-zeros (nnz)	6,940,000	67,303,269
Matrix entries	Scalars	$(3, 3)$ dense blocks

Table 8.2 – Number of edges in DAG.

Matrix	ILU	Edges
CUBE_100	ILU(0)	2,970,000
	ILU(1)	5,910,300
	ILU(2)	10,761,498
SPE10	ILU(0)	2,970,000
	ILU(1)	5,910,300
	ILU(2)	11,357,865

computationally heavier than Cube_100 tasks. Each case generates approximately 1 million tasks. The test protocol is the following: each test is run three times, the final retained result is computed as the average of the three measured results. For each test, we collect three different timings: the factorization time, the triangular solve time, and the aggregation time. We perform the tests first on a single socket, and then on two sockets. The level k of ILU(k) preconditioner determines the level of fill of the matrix. In others words, the number of computations per line and the number of dependences between tasks grow up with a larger k value. We test three levels of ILU(k) fill: 0, 1 and 2. Table 8.2 gives the number of edges in the DAG resulting from the factorization for ILU(0), ILU(1) and ILU(2). For the parameterized aggregation heuristics, we select the parameter values leading to the best performance result. In all tables, the best results are represented in bold.

8.3.1 ILU(k) Factorization Step

The first test series is performed on the ILU(k) factorization step on a single 4-core socket (Tables 8.3). With aggregation disabled the task-parallel ILU(0) factorization is always slower than the sequential version, due to the additional cost of task management. The average task duration for CUBE_100 in ILU(0) is only $50\,ns$, while it is $240\,ns$ for SPE10; the management cost of a task is about $500\,ns$ on average. In ILU(2), the average task duration is $600\,ns$ for CUBE_100 and $1.7\,\mu s$ for SPE10. However, these are still not heavy enough for task management to be considered negligible. With aggregation enabled using a `CD(4)` coarsening strategy string, the number of tasks is reduced to 2,500, with an average task duration 400 times larger. In ILU(0) we achieve a moderate speed-up of 2. ILU(2) factorization achieves a better speed-up of 3. The Front algorithm is not as effective as the Depth Front in this test case, because it does not aggregate tasks with contiguous matrix lines, which causes additional cache misses. With two 4-core sockets (Tables 8.4), the parallel ILU(0) again performs slower than sequential execution when aggregation is disabled. With aggregation enabled, the ILU(0) achieves a speed-up of 3. ILU(2) achieves a speed-up of 6.2.

Table 8.3 – ILU(k) factorization step on a single 4-core socket.

Matrix	ILU	Sequential (second)	No agg. (speed-up)	F(32) (speed-up)	CD(4) (speed-up)
CUBE_100	ILU(0)	0.056	0.20	1.06	2.23
	ILU(1)	0.142	0.46	1.54	2.81
	ILU(2)	0.611	1.30	2.58	**3.47**
SPE10	ILU(0)	0.262	0.65	1.89	3.09
	ILU(1)	0.721	1.39	2.30	3.22
	ILU(2)	1.936	1.87	2.19	**3.32**

Table 8.4 – ILU(k) factorization step on two 4-core sockets.

Matrix	ILU	Sequential (second)	No agg. (speed-up)	F(32) (speed-up)	CD(4) (speed-up)
CUBE_100	ILU(0)	0.056	0.28	1.27	2.54
	ILU(1)	0.143	0.65	1.89	3.79
	ILU(2)	0.612	1.47	3.06	**3.91**
SPE10	ILU(0)	0.260	0.97	2.48	3.78
	ILU(1)	0.771	2.24	3.80	5.72
	ILU(2)	2.006	3.37	3.97	**6.21**

8.3.2 Triangular Solve Step

The triangular solve step is itself composed of two parts: a *forward* substitution followed by a *backward* substitution. The DAG of the forward substitution is identical to the DAG of the factorization step mentioned in the previous section. Thus we reuse the same coarse DAG. In our test cases, the DAG of the backward substitution part is the transpose of the DAG of the forward substitution. Here again, the factorization coarse DAG can thus straightforwardly be reused. In total we have twice more tasks in triangular solve than in factorization. The weight of operations done in triangular solve elementary tasks is lighter than their factorization task counterparts. Tests on a single 4-core socket (Tables 8.5) show that the parallel triangular solve is always slower than the sequential version with task aggregation disabled. With aggregation enabled, we obtain a speedup of 2. On two 4-core sockets (Tables 8.6), with task aggregation disabled, only the SPE10 test achieves a speedup greater than 1. With aggregation enabled, a speedup of 2.52 is achieved on ILU(0) and a speedup of 4.14 is achieved on ILU(2).

8.3.3 Aggregation Overhead

The fine-grain DAGs generated from the test cases amount to about 1 million nodes for each matrix. The number of edges increases when the parameter k of ILU(k) preconditioner increases (see Table 8.2). This aggregation step is performed only once for each matrix. Then, the coarsened DAG is reused as long as the sparse pattern of the matrix is unchanged. In a classical numerical simulation, the mesh on which the equations are discretized does not change through the simulation, only the coefficients of the linear system are changing. This means that the aggregation step is typically done only once while the factorization and

Table 8.5 – Triangular Solve step on a single 4-core socket.

Matrix	ILU	Sequential (second)	No agg. (speed-up)	F(32) (speed-up)	CD(4) (speed-up)
CUBE_100	ILU(0)	0.092	0.23	0.88	1.90
	ILU(1)	0.117	0.27	0.82	1.97
	ILU(2)	0.163	0.34	0.92	**2.05**
SPE10	ILU(0)	0.219	0.40	1.08	1.93
	ILU(1)	0.353	0.62	1.37	2.22
	ILU(2)	0.554	0.85	1.58	**2.39**

Table 8.6 – Triangular Solve step on two 4-core sockets.

Matrix	ILU	Sequential (second)	No agg. (speed-up)	F(32) (speed-up)	CD(4) (speed-up)
CUBE_100	ILU(0)	0.092	0.27	1.27	2.44
	ILU(1)	0.123	0.35	1.54	2.82
	ILU(2)	0.174	0.45	1.40	2.98
SPE10	ILU(0)	0.219	0.53	1.63	**2.52**
	ILU(1)	0.408	0.96	2.39	3.77
	ILU(2)	0.658	1.38	2.79	**4.14**

triangular solves are called a large number of times (several thousand of times). The time spent in the task aggregation (Table 8.7) is then usually negligible compared to the time spent in the factorizations and triangular solves. On our platform, on average, applying *CD(4)* Coarsening String takes 1.5 s on a DAG with 1 million of nodes and applying *D(400)* takes 3.5 s. A *CD(4)* aggregation saves on average 0.22 s per factorization and 0.31 s per triangular solve compared to not doing aggregation. So after only 3 combinations of factorization and triangular solve, the aggregation become profitable. The *D(400)* aggregation saves on average 0.28 s per factorization and 0.49 s per triangular solve, thus after 5 combinations of factorization and triangular solve, the aggregation becomes profitable.

Table 8.7 – Aggregation overhead.

Matrix	ILU	F(32) (second)	D(400) (second)	CD(4) (second)
CUBE_100	ILU(0)	1.534	2.945	0.908
	ILU(1)	1.467	3.032	1.119
	ILU(2)	1.871	3.705	1.315
SPE10	ILU(0)	1.519	3.239	1.217
	ILU(1)	1.542	3.344	1.527
	ILU(2)	2.019	4.042	1.918

8.4 Discussion

Beyond this initial design are several possible directions for improvements and future works. One important first direction to explore is related to the decision making in selecting aggregation policies and parameters. A practicable way to help in the decision making about aggregation could be to use static compiler analysis or leverage tools such as MAQAO (see Section 9) to estimate the computational weight of tasks, and determine the number of tasks to aggregate to reach the profitable task weight threshold. A compiler could also help in applying loop unrolling techniques to coarsen tasks at compile time towards the target weight. Also, a data reference analysis could help in identifying affinity relationships between tasks and deciding about which strategy to select.

A second direction of improvement concerns the set of operators offered by the framework. The current set of operators is a set of "hard decision" operators, as the initial objective of this work was to make task parallelism worthwhile for mostly regular lightweight pieces of computations. A step beyond would be to add conditional statements to the aggregation strategies, to adapt to less regular codes, and to be able to match a larger set of patterns. Weight-oriented conditionals could also be envisioned, though hard-wiring absolute weight constants in the strategies should be avoided to preserve portability. Thus, the runtime system could supply a reference task weight at run-time, and let the applicative code express weight-oriented conditionals relative to this reference computational weight. The StarPU runtime system, for instance, provides means to estimate the minimum weight of tasks to target, in order to scale (e.g. for the scheduling benefit/cost ratio to become profitable), for a given platform and a selected scheduling policy; however, this information is currently only made available to the programmer through some micro-benchmark test programs in the distribution. An alternative could be to run the testing code as part of the calibration process at initialization time, to obtain the target reference weight automatically.

Regarding portability, a third direction would be to target other task-based runtime systems beyond Intel TBB and OpenMP back-ends. This should likely be straightforward as the Taggre environment only has moderate requirements, basically any task parallel model —though with recursive-enabled task models, the recursivity already offers a task granularity control means besides aggregation—. In the case of OpenMP, the Taggre environment could also act in cooperation with some dedicated OpenMP extensions, in the same line as what the `omp collapse` clause does for loops.

Alternatives to the Taggre aggregation technique exist also. First, it is possible, as noted in Section 6.2, to use nesting interoperability, that is, to use a lightweight runtime system to manage fine-grained task parallelism *inside* coarse tasks scheduled by a more heavyweight runtime system. Also, the OmpSs development team explored fine-grained task dependences between nested tasks of one task and nested tasks of another task, thus reconciling the recursive tasks model to control the grain with the dependent task model to avoid heavy synchronizations between outermost tasks. Other possibilities include task-oriented evolutions of the BubbleSched concept developed as part of Samuel Thibault's PhD Thesis [164, 165] to enable having multiple selectable views of a same kernel, either as a single task or as a task graph, as explored by Pierre-André Wacrenier, deferring the decision at run-time, in a similar way as a Just-In-Time compiler.

8.5 Conclusion on Tuning through Task Granularity

In this chapter, we studied the TAGGRE framework —developed as part of the PhD work of Corentin Rossignon— for applying coarsening patterns to a graph of tasks too light to be scheduled individually in a profitable way. While not automatic, the framework enables the applications of patterns and sequences of patterns in a simple way, and allows the programmer to preserve the natural expression of the algorithm. We then discussed several possible directions to extend the framework.

Next chapter will now present a performance tuning methodology based on the automatic suggestion of source code transformations to improve the vectorization of programs, from the analysis of binary codes with the MAQAO framework.

Chapter 9

Tuning through Code Transformation

In this chapter, we study performance tuning techniques through code transformation, and define an approach to guide programmers in the process of applying such transformations based on an analysis of compiled applicative code binaries. The study more specifically explores means to improve the use of data-parallel processor instruction sets, from the point of view of data structure layouts.

9.1 Context

For modern processor architectures, employing Single Instruction, Multiple Data (SIMD) instructions and using them efficiently is essential in order to reach high levels of performance. With the increase of vector width —up to 16 floats in instruction sets such as the INTEL AVX-512, for instance— SIMD instructions are readily available performance multipliers. Several options are given to the application developer in order to vectorize a code: explicit vectorization through assembly instructions, intrinsics routines, C++ wrapper libraries such as MIPP [44], GCC vector extensions or other language extensions (such as INTEL SPMD Program compiler for instance [129]), or implicitly through the automatic vectorization support of the compiler. Explicit vectorization, however, is complex and time consuming; assembly and intrinsics-based approaches also are not portable. Consequently, the vectorization effort for most applications is delegated to the compiler, which may fail to meet the programmer expectations, depending on the code structure and complexity. Indeed, a conservative dependence analysis, an incomplete static information about the control-flow or an unsupported data layout are among the main reasons why the compiler may not generate vector code, even though the source code actually happens to be vectorizable.

Therefore, determining whether the code is vectorizable —independently of compiler limitations— as well as pinpointing the issues that may hinder vectorization together with the possible code transformation remedies are critical capabilities for the developer. We give an overview of the technique we developed in MAQAO [23] —the Modular Assembly Quality Analyzer and Optimizer, a binary code analysis, instrumentation and patching tool— for detecting such SIMD [1] opportunities [15, 89, 18] and hinting at possible performance improvement through code transformations, within the context of the PhD Thesis of Christopher Haine [88], for which I was co-supervisor.

1. This topic leads to the interchangeable use of terms such as *SIMDizability* and *SIMDization*, and more generally *vectorizability* and *vectorization* in the following.

9.1.1 MAQAO

MAQAO is a performance tuning tool [23] that analyzes the binary code of applications, written in C/C++ or Fortran. It was initially developed at the University of Versailles in France, and has partly been further developed at INRIA in Bordeaux as well, since around 2010. While primarily focused on INTEL processors (x86-64, XEON PHI, and even ITANIUM), it was ported on the ARM architecture as part of FP7 MONT-BLANC 2 European Project work.

The core libraries of MAQAO enable the fundamental operations to disassemble a binary file, modify its assembly instructions (operation designated as "patching"), and re-assemble the modified binary. Possible instruction modifications include moving/inserting/suppressing code blocks, inserting function calls, or operations designated as "instrumentation" where some sequences of instructions are both moved and modified, such that a function call is made every time such instructions are called, before executing the instrumented instructions themselves. Core services also include foundational capabilities such as building call graphs, control flow graphs and instruction level dependence graphs from the disassembled binary functions.

A set of LUA-to-C wrapper routines implement the transition between the C core and the LUA services. These routines enable to manipulate relevant assembly objects such as binary files, functions, basic instruction blocks, individual instructions, instruction operands, loops and labels.

On top of those transitional routines, special-purpose processing modules can be implemented in LUA to perform high level operations such as analysis, tracing and hinting. In particular, tracing involves *instrumenting* every memory reference in the studied kernel with calls to the memory access accounting routine in the MAQAO Trace Library (MTL), which monitors every memory address referenced within the kernel routine during an execution, and generates a *compressed trace* of the memory references. The trace can then be further processed to output memory access patterns as human readable algebraic expressions.

MAQAO's instrumentation is able to capture any value in the code, and in particular can be used to trace memory accesses, count loop iterations, capture function parameters. Compared to PIN [113], a tool with similar functionalities, MAQAO performs only static analysis of binaries and static rewriting (from binary to binary). PIN, on the contrary, dynamically rewrites binary codes while they execute, and performs analysis on the fly. As the static analysis of MAQAO is offline, as well as the instrumentation process, the overall cost for analyzing a binary with MAQAO is much smaller than with PIN.

Executing the resulting instrumented kernel produces a trace file containing a compressed representation of the target addresses of each memory access instruction encountered during execution. For each instruction instrumented, the flow of addresses captured is compressed on-the-fly using a lossless algorithm named *Nested Loop Recognition* (NLR), designed by Ketterlin and Clauss [104]. The trace is stored using a compact text file format, not meant to be read as-is by the programmer, as shown below.

```
D 1 E 10 R 1 I 1 K 0 V  L P 39 L P 259 0 T 1 P 1262052 0 8 0 N N N I 2 K 0 V  L P 39 L P 259 0 T 1 P
3439840 0 8 0 N N N I 3 K 0 V  L P 39 L P 259 0 T 1 P 3439852 0 8 0 N N N I 4 K 0 V  L P 39 T 1 P
3199305884 0 N N I 5 K 0 V  L P 39 T 1 P 3199305888 0 N N I 6 K 0 V  L P 39 T 1 P 3199305892 0 N N I
7 K 0 V  L P 39 T 1 P 3199305896 0 N N I 8 K 0 V  L P 39 L P 259 0 T 1 P 1262052 0 8 0 N N N I 9 K 0
V  L P 39 L P 259 0 T 1 P 3439840 0 8 0 N N N I 10 K 0 V  L P 39 L P 259 0 T 1 P 3439852 0 8 0 N N N
```

This intermediate format can however be processed by MAQAO's Memory module to produce a human-readable version of the NLR trace. The command below produces the

human-readable form of the trace from the compact trace and from the companion LUA meta-data file generated during the instrumentation step.

```
$ maqao memory -d -t=<COMPACT_TRACE> -meta=<META_LUA_FILE> -tp=<TEMP_DIR>
```

This human-readable version shows, for each instrumented instruction, the corresponding thread id, the id of the loop containing the instruction as assigned by MAQAO, and the "instrumented instruction" id also incrementally assigned by MAQAO. Multiple instances may exist when multiple threads invoke the instruction.

```
Info: ###############
Info: ## Volume for tid = 0 loopid = 21 iid = 0
Info: ###############
Info: Instance 0
Info: ###############
for i0 = 0 to 39
  for i1 = 0 to 259
    val 0x1341e4 + 8*i1

 [...]
```

Then, following the identification information, the successive values captured by the instrumentation are represented as a pseudo source-code with loops and expressions. The expressions describe the memory addresses that have been accessed, and depend on the surrounding loop counters and loop bounds, as detected by the NLR algorithm. The portion of trace above corresponds to the Array `b` reference in the basic loop nest below, extracted from routine `s111` of the TSVC benchmark suite [115, 41]. Array elements in this example are 32-bit single precision floating point values. Thus, the `8*i1` expression in the trace expresses a 2-element stride array traversal, corresponding to the innermost loop `i1` index step value in the source code. The index `i0` is not referenced in the expression, meaning the outer loop is a repetition loop.

```
1 /* TSVC s111 kernel */
2 for (int nl = 0; nl < 2*ntimes; nl++) {
3       for (int i = 1; i < LEN; i += 2) {
4             a[i] = a[i - 1] + b[i];
5       }
6 }
```

Trace expressions can, however, only depend linearly on the loop counters, and loop bounds only depend linearly on enclosing loops' counters. The memory addresses referenced by an expression form an union of polytopes. Thus, the method captures the memory working-set as well as a schedule for the memory accesses. Figure 9.1 shows another example of traces for a simple code, assuming elements of the arrays are 4-byte long floats.

Three important features for these traces are used in this work: (i) Regular strides are captured. This is important for SIMD optimization, since this will decide whether data layout restructuring is needed or not. In Figure 9.1.(b), the stride is 4, meaning data of array C is contiguously accessed in this write. For the array B, the stride of 8 shows that one float out of two is read, data is not contiguously accessed. (ii) Regular streams are fully traced, in a compact form. This enables the computation of dependence distances. (iii) For multi-dimensional data, memory expressions provide spatial locality information through the ordering of the strides. This can be used in order to propose loop restructuring hints.

For irregular patterns, new loops are created, possibly leading to a trace with no compression if no regularity is found.

```
for(i=1; i<100; i++)
  C[i] = C[i - 1] + B[2 * i];
```

(a) Code example

```
for i0 = 0 to 98
   write 0x2ba1a3bd442c + 4 * i0
```

(b) Compressed trace for `C[i]`

```
for i0 = 0 to 98
   read 0x2ba1a3bd4428 + 4 * i0
```

(c) Compressed trace for `C[i-1]`

```
for i0 = 0 to 98
   read 0x2ba1a4000000 + 8 * i0
```

(d) Compressed trace for `B[2 * i]`

Figure 9.1 – Example of trace compression using NLR. For the code in (a), one compressed trace per memory access is produced, (b), (c) and (d).

9.1.2 Hybrid Static/Dynamic Dependence Graph

The dependence analysis we propose is a combination of a static dependence analysis, for registers, and dynamic dependence analysis for memory dependences. The static dependence analysis on registers is already implemented in MAQAO and corresponds to an SSA analysis. Memory dependences are obtained by tracing with MAQAO all memory accesses within the innermost loops, and then computing dependence distances.

Static, Register-Based Dependence Graph. The dependence graph on registers is produced from a SSA (static single assignment form [59]) analysis, proposed by MAQAO. Besides, MAQAO handles special cases for dependences:

— xor instructions, applied twice to the same register, set the value of this register to 0. While the operation is reading a register, this is not considered as a read access.

— Some SIMD instructions can operate on the lower or higher part only of a SIMD register. Operations that operate on different parts of a register are not considered in dependence.

In addition to the existing analysis, we tag all dependences where a register is read for an address computation. The graph is then partitioned according to these edges (e.g. cutting the graph through these edges), usually in two parts: Instructions preceding these edges are address computation instructions —such as index computation, update of address registers— while instructions after these edges are actual computation —memory accesses, floating point operations, etc.— for which SIMDization may be applied. When an indirection occurs, the dependence graph has a path with two tagged edges and can therefore be partitioned into three or more subgraphs. The partition of instructions following all tagged edges is said to be the *computational part of the graph*, while the other instructions are part of the *address computation part* of the graph. In the rare cases where it is not possible to cut the graph following tagged edges, we assume there is no computational part.

Dynamic Dependence Graph. Dynamic dependences are essential to capture what the compiler may have missed, concerning the control flow or the way data structures are indexed. The dynamic dependence graph is built from the memory trace for each read and write instructions in innermost loops.

Algorithm 1 describes how dependence distances are computed. $w.trace$ denotes the trace captured for an instruction w. For each couple of read and write accesses in a loop, we first perform an interval test, based on their trace (line 2), and then compute a *dependence distance*. The dependence distance is defined as the number of loop iterations between two instructions accessing the same memory location. When two traces have the same loop structure, the

subtraction between the traces (line 4) subtracts the expressions that are at the same position in the trace. If the result is not the same constant value for all subtractions, then $*$ is returned, otherwise the constant value is returned. The special $*$ dependence distance notation between two instructions denotes the fact that their dependence distance is not constant during the execution of the program. Note that only uniform dependences are captured this way, but as far as SIMDization is concerned this captures all vectorization opportunities that do not require non-local code transformation.

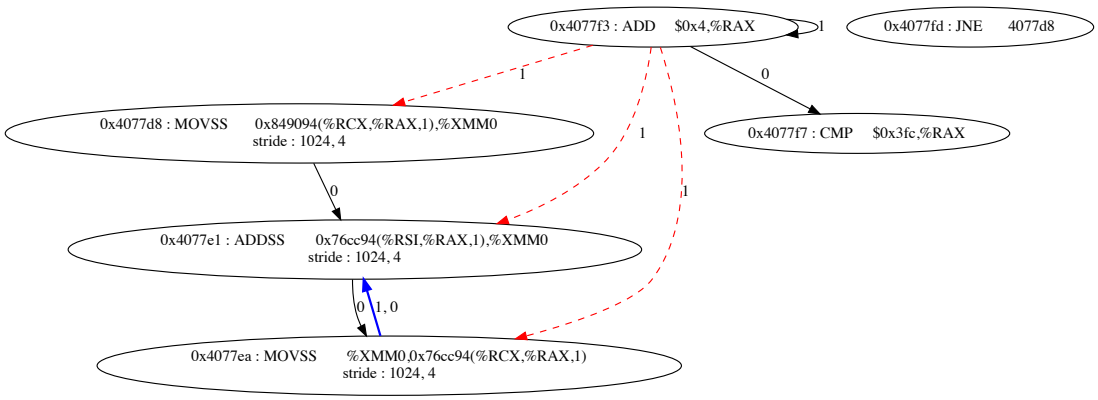
Algorithm 1: Dynamic dependence computation for an innermost loop L.

```
1  for w, a write and r, a read in L do
2  |  if w.trace ∩ r.trace ≠ ∅ then
3  |  |  if loops of w.trace = loops of r.trace then
4  |  |  |  return r.trace − w.trace;
5  |  |  else
6  |  |  |  return ∗;
7  |  |  end
8  |  else
9  |  |  return 0 ;
10 |  end
11 end
```

For the example in Figures 9.1.b and 9.1.c, both traces have the same structure, the same strides, and the difference between the read and the write addresses is an offset of -4. Then we evaluate how this difference can be compensated by a variation in the loop indices (here $i0$). We find a unique solution within the loop bounds, 1, and this shows that the dependence distance between the write and the read is 1. In the general case, finding the vector of iteration counters that compensate for the offset between the read and the write leads to a multi-dimensional dependence vector. Only read-after-write dependences are evaluated, and the sequential order of the assembly code is used to compare relative positions for reads and writes. Note that all distances for register dependences correspond in this case to innermost loop carried dependences. Figure 9.2 presents the s2233 function from TSVC and its dependence graph, combining both the static and dynamic graphs. The nodes each represent an assembly instruction, along with its strides when it is a memory access. The dashed edges in red represent dependences for registers used in address computation. Cutting the graph along these edges separates the computational part (left nodes) from the address computation and control part. The bold blue edge, labelled with $1, 0$, represents the memory dependence corresponding to array `bb` directly computed from the trace. The strides denoted on the edges have two values: 1024 and 4. The first one corresponds to the stride for the innermost loop j, and the second to the i loop. This shows that here, none of the accesses have good spatial locality.

9.1.3 SIMDization Analysis

The SIMDization analysis is performed in two steps. First, we determine whether the code has a parallelism compatible with SIMDization, independently of any data layout or control limitations, such as large stride. Then, we refine the analysis to detect special cases and guide the programmer accordingly towards enabling and improving SIMDizability.




```
1 for (int i = 1; i < LEN2; i++) {
2   for (int j = 1; j < LEN2; j++) {
3     bb[j][i] = bb[j][i-1] + cc[j][i];
4 }
```

Figure 9.2 – Code and dependence graph for one loop of function s2233 in TSVC benchmark.

Vectorizable Dependence Graph. The dependence graph (static and dynamic) is first partitioned according to address computation edges, as described previously. The graph is said vectorizable if one of the three conditions applies to the computational part of the graph:

— There is no cycle.
— There is a cycle, with a cumulative weight greater than the width of SIMD vectors.
— There is a cycle, with a cumulative weight smaller than the width of SIMD vectors, *and* the instructions of the cycle all are of one of the following types: `add`, `mul`, `max`, `min`. The cycle then corresponds to a reduction.

A code with a vectorizable dependence graph may yet require transformations in order to be actually SIMDizable. This is detailed in the following section.

Code Transformation Hints. From the dependence graph, the stride expressions, and the control flow graph analysis, our method generates the following hints.

Data alignment: When the graph is vectorizable without cycle, misaligned data is detected by comparing the starting address of all memory streams with the width of SIMD vectors. In the simpler case, the user can either change memory allocation of heap-allocated data structures, or use pragmas for aligning stack-allocated data. When for instance $A[i]$ and $A[i+1]$ occur in the same code, one of the two accesses is misaligned. This would require shuffle instructions, or unaligned loads/stores whenever they exist.

Rescheduling: When the graph is vectorizable without cycle, there may still exist some dependences with non-null distance. Due to the fact that the analysis is performed on assembly code, this may require some modifications at the source code such as some rescheduling of loads/stores and computations, splitting some larger instructions. A template of the vector code is generated by our tool (an example is given in the following section), giving a correct instruction schedule after SIMDization.

Loop transformations: Loop interchange is proposed when all accesses within the loop have

a large innermost loop stride, and another loop counter in the expression corresponding to the same outer loop has a stride 4 (for 32-bit floats and integers). Interchanging these two loops would result in better locality and enable SIMDization.

Loop reversal: Traces with negative stride expressions lead to this hint. Note that if other reads for instance have a positive stride, the reversal is not beneficial any more.

Data reshaping required: This is a fallback hint for large innermost loop strides, and for codes with indirections (detected on the static dependence graph). On the SANDY-BRIDGE and XEON PHI architectures, instructions for loading or storing non-consecutive elements into/from SIMD vector have been added to the ISA (GATHER on XEON PHI and SANDY-BRIDGE and SCATTER on XEON PHI). The use of these instructions, through assembly code or intrinsics, is an alternative to data reshaping.

Versioning required: The static analysis on the code may lead to a different conclusion than the trace analysis. For instance, the trace may find a regular stride for a memory stream whereas statically, this stream results from an indirection, or depends on a parameter. Similarly, the dynamic control flow (e.g. the real path taken) may be a subset of a more complex static control flow. In these cases, the trace may have captured only one behavior of the code, for a particular input. The vectorization may only be possible in this case through the versioning of the loop, depending on the values of the array, of a parameter.

Idiom Recognition: When the code is vectorizable, with 0 dependence distances or with reductions, the dependence graph can match a predefined dependence graph representing a well known computation. The shape of the dependence graph and the instructions themselves are matched with the predefined graphs. In this case, the user can call a library function instead of trying to vectorize the actual code. The predefined functions considered so far are: dot product, daxpy, copy, sparse copy (copy with an indirection either in the load or in the store), but more complex functions can be added with ease.

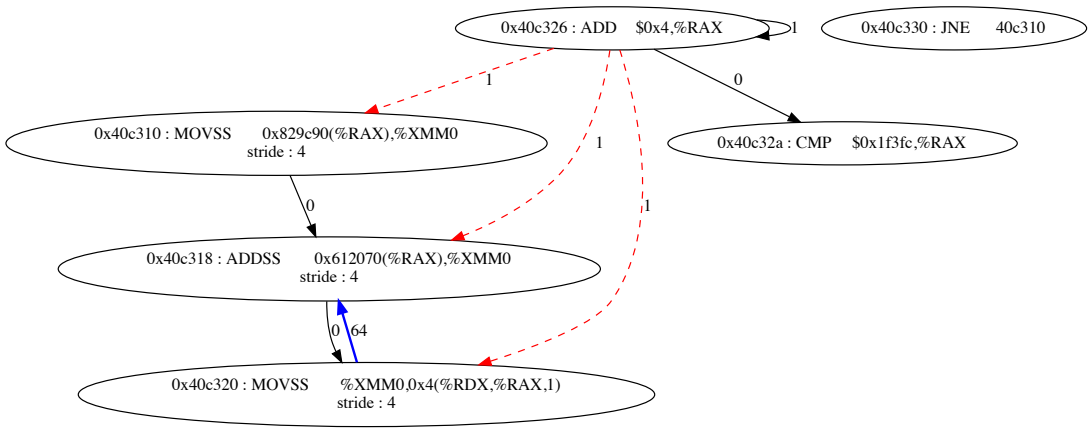
9.1.4 Example on the TSVC Benchmark

The TSVC benchmark suite [115, 41] has 151 codes with small loops, illustrating different vectorization difficulties. We first present the output generated by our method on one example and then show aggregated results for all TSVC benchmarks.

Figure 9.3 presents a code with an alias between two arrays. This kind of alias can hinder automatic vectorization and only dynamic dependence analysis or possibly inter-procedural alias and points-to analyses are able to cope with such situation. Here, the dynamic dependence analysis shows that the dependence distance between the write of `x[i+1]` and the read of `array[i]` is 64 iterations. Thus vectorization is possible as long as the vector width is < 64. The output generated by our method is the following:

```
Loop at lines 4443-4444 of tsc.c:
  vectorizable
  contiguous data
  code template:
     load (i:i+4)           line 4444
     load (i:i+4) and add   line 4444
     store (i+64:i+68)      line 4444
```

The source line and name of the file are provided by MAQAO and extracted from debug information. User-friendly names are associated for most frequent instructions found in the computational part of TSVC. The dependence between the store and load is represented by the dependence vector on the indices.




```
1 for (int i = 0; i < LEN - 1; i++)
2   xx[i+1] = array[i] + a[i];
```

Figure 9.3 – Code and dependence graph for the loop in function s424 of TSVC benchmark. `array` is aliased with xx with an offset of 63 floats.

Table 9.1 presents the overall SIMDization opportunity hints generated by the analysis on all TSVC benchmarks. Out of the 151 functions to analyze, 123 are detected as SIMDizable, with 0, 1 or more hints provided by MAQAO. If we focus on the codes that are said vectorizable and are not vectorized by GCC, for 23 of them, speed ups greater than 2 are obtained through hand vectorization, compared to GCC generated codes.

Table 9.1 – MAQAO vs GCC and INTEL ICC compilers.

	Maqao	**GCC**	**ICC**
Detected vectorizable cases	123	46	104
Corresponding MAQAO *hint*			
- Reduction	30	15	24
- Idiom	8	3	7
- Data alignment issue	11	4	4
- Code transformation	53	6	39
- Loop interchange or data transpose	9	4	7
- Rescheduling	9	1	1
- Control	23	0	17

These hints provide the user with possible strategies to remove SIMDization hurdles, such as code transformations or data restructuring. However, this initial work conducted a qualitative analysis only, thus lacking an estimation in the transformation gains. Therefore, we then extended our technique to assess potential data restructuring gains in a quantified manner [89, 18].

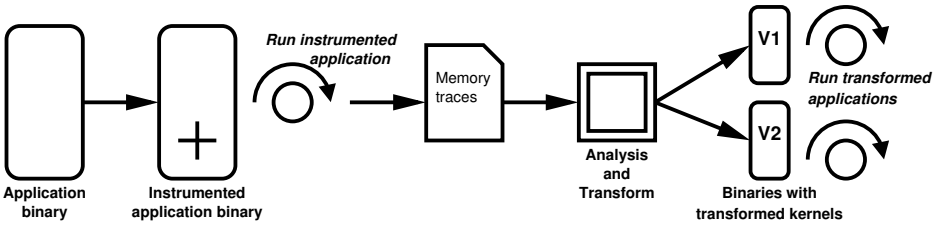


Figure 9.4 – Data-layout transformation evaluation big picture.

9.2 Transformation Hints Assessments

This section deals with the quantified part of the user feedback we provide. The idea is to estimate the potential speedup of transformations in order to help the user make a choice for data restructuring.

9.2.1 Principle of Mock-up Evaluation

We propose an evaluation methodology that explores a set of different layout transformations. Because these transformations are based on the values collected from memory traces, the generated transformed codes are in general not semantically equivalent to the initial code, outside of the trace. However they can serve as performance "mock-ups".

Our proposed methodology is shown on Figure 9.4 through the following steps: 1. The original application binary is instrumented with MAQAO in order to trace the targeted hotspot region. 2. The instrumented binary is then run to generate a trace of memory references. This trace consists of the sequence of addresses accessed for each read/write operation and for each thread. 3. The application binary together with the generated memory traces are analyzed in order to propose possible transformation strategies to improve SIMDizability. The analysis determines which arrays are accessed and how they are accessed, then the transformation changes this data layout in order to enhance spatial locality. Transformations are array contraction, transposition and transformation from Arrays of Structures (AoS) into Structures of Arrays (SoA). A SIMDization step is performed if possible, translating instructions into their SIMD counterparts. Finally, a transformed code is generated for each such strategy. Since these codes are generated with transformations relying on trace information, they may not be semantically equivalent to the initial code. They are called *mock-ups*. 4. Each transformed mock-up is then run within the context of its host application. Its performance is measured to assess the relevance of the corresponding strategy.

The idea is to measure possible performance gains of the application by executing the mock-ups. To preserve the application execution conditions, the mock-up is executed in the context of the application. A checkpoint/restart technique is used for this objective: assuming the user knows the hotspot of the application, the original binary code is patched with a checkpoint right before the hotspot and then run until the checkpoint is reached. This checkpoint generates an execution context, used for capturing the trace and running/evaluating the mock-ups. The binary code is instrumented in order to collect the memory trace and restarted from this context. Then several layout transformations are applied on the initial code, generating new versions of the code that are restarted from the same context. As the checkpoint/restart mechanism preserves the memory addresses in use, the addresses and sizes of layouts captured in the trace can be reused in the mock-up codes. We rely on this

property for generating data layout copies and the transformed codes. Our approach does not preserve the hotspot cache state however. Cache warm-up may be a solution to this issue, but goes beyond the scope of this paper. The execution of mock-ups is stopped when the control leaves the hotspot and the timing is measured at this point. For checkpoint/restart, we resort to the BLCR library [90].

Automatic Mock-up Generation Technique Mock-ups are generated at compile time, as library functions. A mock-up corresponds to the initial hotspot, with different memory accesses and their address computation. The rest of the computation itself corresponds to the original code.

Before executing the mock-up, the data layout has to be created and data copied. This copy-in operation is guided by the trace information. The objective is to optimize the hotspot performance, and to push away the copies from the kernel to minimize their impact, avoiding cache pollution due to the copy itself. We choose to move the copy up to the beginning of the function when applicable, the limit being the last write on the array we want to restructure. This is determined automatically by trace inspection.

The sequence of transformation rules applied to the initial layout corresponds also to transformations on the iterators of these structures. The copy codes are simple loops changing one layout, with one iterator, into another. For the indexation of data in the computation code, the control is kept unchanged. New scalar iterators are created in order to map the previous index to the new index. For this, the trace provides for each individual assembly instruction the sequence of addresses accessed. This sequence of indices is transformed into a sequence of new indices, of the new layout.

The binary code is parsed with the MAQAO tool, and the modified code of the mock-up is generated in a C file, using inline assembly. The advantage of this approach is to rely on the compiler to perform an optimized register allocation for all the new induction variables added for indexing, and also to remove dead code. For instance, the loads corresponding to the indirection are removed when re-indexing the data structure in a simpler way. The code generated is only valid within the scope of the values collected by the trace.

9.2.2 Combining Layout Restructuring with SIMDization

Data restructuring is a SIMDization-enabling transformation, since data can be laid out contiguously to fill a vector. We perform SIMDization whenever dependences allow it. From the trace analysis, we build a dependence graph that determines whether some arrays can be vectorized. We rely on MAQAO for this analysis [15], as well as for the detection of loop structures and for loop counters. The generated vectorized loop has a shorter loop trip count by a factor equal to the architecture vector size. This loop trip count is retrieved from the memory traces. All instructions involving the initial data structure have to be replaced by their vectorized counterpart, including load and stores. Some compiler optimizations can be untangled, such as partial loads that are replaced by a single packed load operation. Reductions are detected through dependence graph analysis, and are replaced using horizontal operations. We detect read-only arrays or constants and unpack them. However, the SIMDization step from binary code to assembly code (assembly inline) is still rather fragile.

9.3 Discussion

Thus, a lot of work remains to make the prototype more robust, due to the potentially limited view given by the trace, depending on its variability across multiple runs, and due to the difficulty to assess the impact of a local kernel transformation on the applicative code as a whole. Moreover, there is no guarantee offered that the restructured code mock-ups remain semantically equivalent. Yet, it would be much valuable to offer some confidence assessments that the equivalence is likely preserved, especially in the absence of data related branching.

Going further, the scope of the Layout Restructuring chain could be broaden to generate accelerator mock-ups. Developing code for accelerators indeed remains challenging and expensive, and due to the nature of accelerator processors, porting a CPU kernel onto an accelerator is not necessarily worthwhile if the kernel is not sufficiently regular in its computational flow, or not heavy enough to justify an offload operation. Thus, generating an accelerator mock-up could help in guiding the decision early to develop, or not, an accelerator port of a kernel with a rough overview of the possible benefits. Likewise, the chain could help in the process of "taskification", e.g. to transform a kernel in a task-parallel oriented code, to assess the worthiness in terms of task grain size (see Section 8.4).

The model itself could be enriched also. As part of the FP7 MONT-BLANC 2 European Project, we experimented with the interfacing of MAQAO with the BOAST [170, 171] kernel tuning framework. BOAST has the ability to generate many variants from a unique, parameterized kernel source code. We designed a prototype mechanism to have MAQAO instrument and analyse BOAST-generated kernel binaries, combining the output report with annotations created by BOAST during the kernel generation process. The interest of this cooperation is twofold: on the one side, to transmit information to MAQAO about the parameters used by BOAST in generating a kernel variant; and on the other side, to alter BOAST parameters according to MAQAO output. The couple BOAST + MAQAO could then be used to study the evolution of the trace across multiple runs or in reaction to some parameter changes, to discriminate invariant portions from data-sensitive or parameter-sensitive ones.

To enhance the model additionally, one could also leverage hardware counters available in modern processors to get more information on the performance and impact of the original kernel compared to mock-ups, for instance to detect that a suggested transformation improves SIMDization but unexpectedly causes a higher cache miss rate as well. Another possibility would be to combine the information obtained through the chain with information obtained from performance models such as those built by STARPU, relating performance metrics to some input parameter such as the input data size.

9.4 Conclusion on Tuning through Code Transformation

In this chapter, we studied the methodology designed as part of the PhD work of Christopher Haine and the work conducted within the European project FP7 MONT-BLANC 2, to characterize issues in compiler-generated binary codes likely to be hindrances to the compiler auto-vectorization process, and to suggest transformations to apply on the source code to remove such hindrances. The methodology builds on the MAQAO assembly code analysis and patching framework to provide the issues detection and characterization, through a process of instrumentation, trace collection, analysis and transformation hints reporting. It also optionally proposes the generation of "mock-ups" of suggested transformations to let the

programmer estimate potential performance gains from such transformations before actually implementing them. We then discussed possible followups in terms of robustness, both regarding the model by itself, and regarding some possible crossbreeding with informations gathered by other tools.

Next chapter will now present a third tuning approach, this time using code generation techniques to specialize the implementation of a whole family of error correction codes algorithms for selected coding instances.

Chapter 10

Tuning through Code Specialization

In this chapter, we investigate the use of code specialization techniques on a family of Error Correction Codes algorithms, the Polar Codes, to optimize their implementation and to enable the use of data parallelism instruction sets on them. The complex, irregular pattern of such algorithms leaves them out of reach from the vectorizing engines of modern general purpose compilers. The study explores the use of code rewriting techniques to implement domain specific optimizations and vectorization, while preserving the genericity of the approach within the Polar Codes family as a whole.

10.1 Context

In 2015, we started a collaboration with the neighbour electronics laboratory IMS from the University of Bordeaux on the topic of the optimization of Error Correction Code (ECC) software decoding. The cooperation proved successful and eventually led to the PhD Thesis of Adrien Cassagne [43] co-hosted by IMS and INRIA, for which I am one of the co-advisors, and to the design of a toolchain for studying, developing and validating ECC algorithms. This toolchain initially started on the specific case of *successive cancellation decoding* for the so-called "Polar" family of ECCs, and was first designated under the acronym P-EDGE (Polar ECC Decoder Generation Environment). It was subsequently renamed AFF3CT, *A Fast Forward Error Correction Toolbox* [46], as it has since been extended and enhanced to support a large number of families of ECC algorithms, in a highly generic and configurable way. We present here some key elements of the initial study [47] we conducted using specialization techniques to optimize the successive cancellation decoding of polar ECC codes, which provides a rich introduction to the specific kind of issues encountered when optimizing ECC codes in general.

Error correction coding —also named channel coding— is a technique that enables the transmission of digital information over an unreliable communication channel, by adding specific patterns of redundancy on the sending side to the user payload data. Using knowledge of such redundancy patterns on the receiving side, the decoding process infers the most likely payload data contents to reconstruct the user information, that may have been altered during the transmission over a noisy channel, such as a wireless connection. In contemporaneous communication systems, hardware digital circuits are in charge of performing the encoding (resp. decoding) of transmitted (resp. received) information. These custom ECC circuits inherently lack flexibility and suffer from long, expensive development cycles. The steadily increasing computing performance improvements of low power processors, such

as commonly found in networking and *Internet of Things* (IoT) devices, make it increasingly more desirable to implement such algorithms in software. Moreover, efficient software implementations of these algorithms are also much needed for running them on high end, high performance processors to shorten the computationally intensive *algorithm validation process*. During such a process, long sequences of information are encoded with the studied algorithm, altered with a controlled random noise, decoded, and compared with the initial sequence to assess and quantify the error correcting power (Note: the error correcting power level of an ECC algorithm is often designated under the term "*performance*" in the field, not to be confused with the computer science "speed-oriented" sense). Indeed, some classes of decoding algorithms can take advantage of modern CPU features such as SIMD units, and even many/multi-cores, making the software approach even more desirable.

10.2 Successive Cancellation Decoding of Polar Codes

Polar Codes were recently proposed in [10]. They can achieve very good error correction performance. However, a very large codelength ($N > 2^{20}$) is required in order to approach the theoretical error correction limit proved by Shannon [148]. The challenge is then to design polar codes decoders able to decode several millions bits frames while guaranteeing a compelling throughput. Let's assume we want to transmit K bits over a noisy communication channel. The encoding process appends $N - K$ parity check bits before the resulting N bits codeword can be transmitted over the channel. On the receiver side, the noisy sequence Y is a vector of N real values each corresponding to *a priori* beliefs on the transmitted bits. These beliefs are in the form of a *Log-Likelihood-Ratio* (LLR). Using the knowledge of the encoding process, the decoder job is to estimate the transmitted N-bit codeword based on a received sequence of N LLRs.

The SC decoding algorithm can be seen as the traversal of a binary tree starting from the root node. For a codelength $N = 2^m$, the corresponding tree thus includes $m + 1$ node layers, indexed from $d = 0$ (root node layer) down to $d = m$ (leaf nodes layers). As the tree is initially full, each layer d contains 2^d nodes, each node of that layer d containing 2^{m-d} LLRs (λ) and 2^{m-d} binary values denoted as *partial sums* (s). At initialization, LLRs received from the channel (Y) are stored in the root node. Then, the decoder performs a pre-order traversal of the tree. When a node is visited in the downward direction, the LLRs of the node are updated. In the upward direction, partial sums are updated. Fig. 10.2b summarizes the computations performed in both directions. The update functions are:

$$\begin{cases} \lambda_c & = & f(\lambda_a, \lambda_b) & = & sign(\lambda_a.\lambda_b).\min(|\lambda_a|, |\lambda_b|) \\ \lambda_c & = & g(\lambda_a, \lambda_b, s) & = & (1-2s)\lambda_a + \lambda_b \\ (s_c, s_d) & = & h(s_a, s_b) & = & (s_a \oplus s_b, s_b). \end{cases} \tag{10.1}$$

The f and g functions both generate a single LLR. The h function provides a couple of partial sums.

Before recursively calling itself on the left node, the algorithm apply the f function, respectively, before calling itself on the right node the g function is applied. At the end (after the recursive call on the right node) the h function is applied. The f and g functions use the LLRs (read-only mode) from the current node n_i in order to produce the new LLR values into respectively left and right n_{i+1} nodes. The h function, in the general case (non-terminal

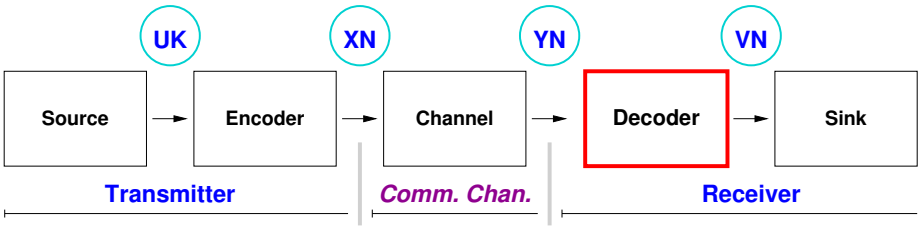


Figure 10.1 – The communication chain.

case), reads the bits from the left and right n_{i+1} nodes in order to update the bit values of the n_i node. For the terminal case, the h function reads the LLRs from itself and decides the bit values.

Leaf nodes are of two kinds: *information bit* nodes and *frozen bit* nodes. When a frozen bit leaf node is reached, its binary value is unconditionally set to zero. Instead, when an information leaf node is reached, its binary value is set according to the *sign* of its LLR (0 if LLR is positive, 1 otherwise). Once every node in the tree has been visited in both directions, the decoder eventually updates partial sums in the root node and the decoding process is terminated. At this point, the decoding result is stored in the root node in the form of a N-bit partial sum vectors.

This decoder algorithm has a number of characteristics of interest regarding optimization. *Generating* decoders able to take advantage of this optimization space is the key for high performance decoders:

— The tree traversal is sequential, but f, g and h are applied element-wise to all elements of the LLR and bits in the nodes and their children. As there is no dependence between computations involving different elements of the same node, these node computations can be parallelized or vectorized (see the *intra-frame* strategy introduced in [84]),
— Frozen bits fully define their leaf values, hence some part of the traversal can be cut and its computation avoided, depending on the location of the frozen bits. More generally, the tree computation can be versioned depending on these bits [6, 143],
— The decoder can be specialized for a particular configuration of frozen bits, as frozen bit locations do not change for many frames,
— Similarly, multiple frames can be decoded concurrently, with parallel or vector code. Such *inter-frame* optimizations can increase the decoding throughput, however at the expense of latency, which is also one important metric of the application [109].

Beside optimizations coming from the computations in the tree, several representations of LLR may lead to different error correction performance. LLR for instance can be represented by floats or integers (fixed point representation), LLR from different frames can be packed together.

Finally, usual code optimizations, such as unrolling or inlining can also be explored. For instance, the recursive structure of the tree computation can be fully flattened, depending on the size of the codelength.

10.3 The P-EDGE/AFF3CT Framework

While the study focusses on the decoding stage, a whole encoding/decoding chain is required for testing and validation purpose. Fig. 10.1 depicts the communication chain of our framework. The chain stages are organized as the following main segments:

The **Transmitter** segment is made of two stages: 1) The source signal generator stage (*Source*) produces the vector of information bits U_K to be transmitted. 2) The polar encoding stage (*Encoder*) inserts parity check redundancy bits into the information vector. For every packet of K information bits, a total of N bits are produced (information+redundancy bits). The resulting N-bit vector (X_N) is transmitted over the communication channel.

The **Communication channel** segment simulates a noisy communication, adding additive white Gaussian noise (e.g. noise with a uniform power across the frequency band, and with a normal time domain distribution) to the frames, producing the real vector Y_N from the binary vector X_N.

The **Receiver** segment is made of two stages: 1) The *Decoder* stage produces a binary vector V_N from Y_N using the algorithm described above. 2) The *Sink* stage eventually compares the K information bits (V_K) in V_N with U_K in order to count the number of remaining binary errors after the decoding is performed. The more effective the error correction code is, the higher number of V_K bits should match U_K bits. Resilient errors may come from 1) inherent limitations in the polar code construction, 2) suboptimal decoding algorithm, 3) high noise power in the communication channel. Moreover, while testing new algorithm implementations or optimizations, an abnormally high error rate can be the sign of a bug.

10.3.1 Specialized Decoder Skeletons, Building Blocks Library

The tree structure at the heart of SC decoders is fully determined by the parameters of a given code instance: the code size, the code rate ($R = K/N$), the position of the frozen bits. The recursive tree traversal code structure and the corresponding tree data structure are challenging to vectorize and to optimize for a compiler. However, all these parameters are statically known at compile time. Our Polar ECC Decoder Generation Environment (P-EDGE) builds on this property to provide a general framework for polar decoder design, generation and optimization.

Beyond the *code parameters*, Polar decoders can be tweaked and optimized in many different orthogonal or loosely coupled ways: *Elementary* type (floating point, fixed point), *Element containers* (array size), *Data layout* (bit packing techniques), *Instruction Set* (x86, ARM), *SIMD* support (scalar, intra-frame or inter-frame processing vectorization), *SIMD instruction set variant* (SSE, AVX, NEON), as well as the set and relative priorities of the *rewriting rules for tree pruning*. Our framework enables to quickly experiment the different combinations of all optimizations. The decoder code thus results from two distinct parts:

— An architecture independent *specialized decoder skeleton* generated by our decoder generator, from a given frozen bits location input. Starting from the naive, recursive expression of the computational tree, we apply successively cuts and specializations on the tree. They are described through a set of rewriting rules that can be customized according to the specificities of the decoder and to the constraints in terms of code size, for instance.

— A library of architecture dependent *elementary computation building blocks*, corresponding to the implementation variants of the f, g and h functions (fixed or floating point

versions, scalar or vector versions, etc.). These blocks do not depend on the frozen bits location and can therefore be used by any specialized skeleton.

10.3.2 Decoder Generation

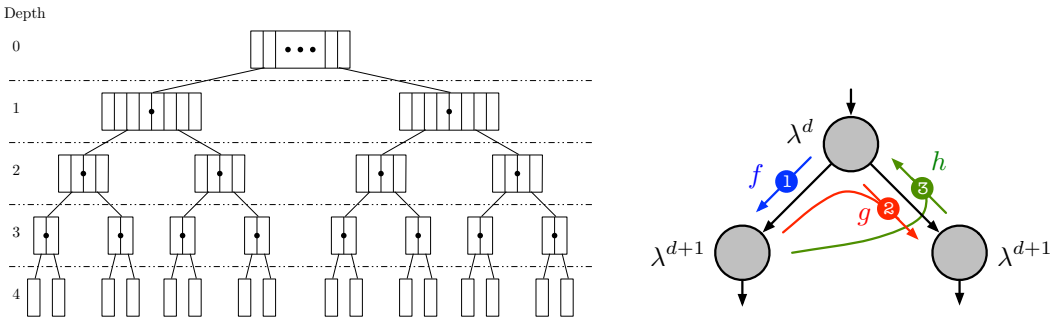


Figure 10.2 – a) Tree layout. b) Per-node downward and upward computations.

The decoder generator first builds the binary tree structure as shown in Fig. 10.2a from the frozen bits location input. Each internal node has a tag indicating the type of processing required at that node (recursive children processing, $f/g/h$ functions to be applied or not). This tag is initially set to *standard*, corresponding to the canonical processing described in Fig. 10.2b.

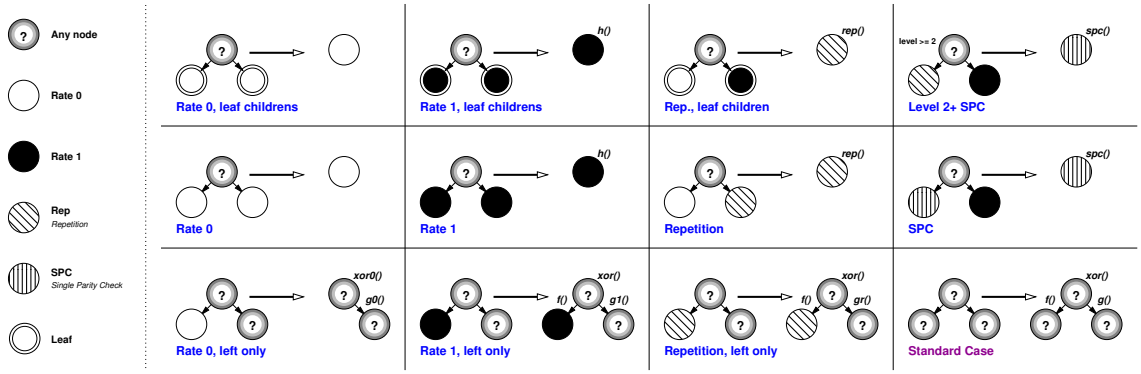
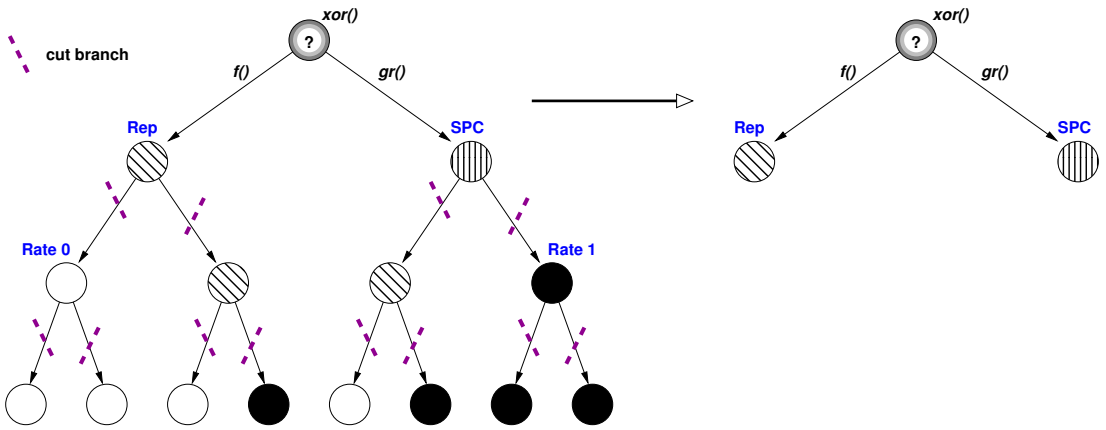


Figure 10.3 – Subtree rewriting rules for processing specialization.

For some sub-tree pattern configurations, the processing to be performed at the root of such sub-trees can be simplified, or even skipped completely, for instance when a node only has two frozen bit leaf children. To exploit such properties, the decoder generator repeatedly applies the set of sub-tree rewriting rules listed in Fig. 10.3 using a depth first traversal to alter the node tags, until no rewriting rule applies anymore.

Each rewriting rule defines a sub-tree pattern *selector*, a new *tag* for the sub-tree root, and the f, g, and h *processing functions* to be applied, simplified or skipped for this node in the resulting decoder. A *null* f (resp. g) function cuts the left (resp. right) child of the node. From an implementation point of view, a rule is defined as a class, with a `match` function, and a set of functions f, g, and h. The current set of rewriting rules can thus easily be enriched with new rules to generate even more specialized versions.

Patterns on the first two rows result in cutting away both children. For instance, the first rule, named *Rate 0, leaf children*, cuts the two frozen bit leaf children of the parent node, and tag it as *Rate 0* (white node). Processing is completely skipped on this node since the values of the bits are unconditionally known. The *Repetition* rules match sub-trees where only the rightmost leaf is black (tag *Rate 1*), the others being frozen bits. In this case, the whole sub-tree is cut and replaced by a simpler processing. Moreover a single, specialized rep function is applied on the node instead of the three functions f, g and h. The third line describes partial cuts and specializations. For instance, the rule "Repetition, left only" specializes the g and h functions to use, but does not prune the recursive children processing.




```
1 void Generated_SC_decoder_N8_K4::decode()
2 {
3   f < R, F, FI, 0, 4, 8, 4>::apply(l );
4   rep<B, R, H, HI, 8, 0, 4>::apply(l, s);
5   gr <B, R, G, GI, 0, 4, 0, 8, 4>::apply(l, s);
6   spc<B, R, H, HI, 8, 4, 4>::apply(l, s);
7   xo <B, X, XI, 0, 4, 0, 4>::apply( s);
8 }
```

Figure 10.4 – Code generation process on a small binary tree ($N = 8$). The tree is cut and the computations are versioned according to the location of the frozen bits. The listing show the final generated code generated.

Rewriting rules are ordered by priority (left to right, then top row to bottom row in Fig. 10.3), thus when several rules match an encountered sub-tree, the highest priority rule is applied. The priority order is chosen such as to favor aggressive computation reducing rules over rules with minor impact, and to ensure confluence by selecting the most specific pattern first. Rules selectors can match on node tags and/or node levels (leaf, specific level, above or below some level). A given rule is applied at most once on a given node.

Finally, once the tree has been fully specialized, the generator performs a second tree traversal pass to output the resulting decoder. An example of such a tree specialization process together with the generator output is shown in Fig. 10.4.

10.3.3 Building Blocks SIMD Support

The main challenge in implementing P-EDGE's architecture dependent building blocks is to offer enough flexibility to support varied data types, data layout and optimization strategies such as intra-frame SIMDization (intra-SIMD) and inter-frame SIMDization (inter-SIMD), without breaking the high level skeleton abstraction. To meet this requirement, our building blocks library heavily relies on generic programming and compile time specialization by the means of C++ templates. Fig. 10.4 gives an example of a generated decoder for $N = 8$, calling template instances of the node functions. The template parameters are defined as follows. `B`: partial sum type; `R`: LLR/λ type; `F/G/H/X`: Scalar standard SC function versions; `FI/GI/HI/XI` SIMD versions. Remaining template parameters are offsets and chunk sizes to control data layout.

SIMD routines are common to both intra-frame SIMD and inter-frame SIMD, and are built on top of the C++ wrapper library MIPP [44] designed by Adrien Cassagne. The **Intra-SIMD** vectorization exploits SIMD units without increasing the decoder latency, since it still processes frames one at a time and thus preserves fine grain frame pipelining. However, at leaf nodes and nearby, too few elements remain to fill SIMD units. For instance, 4-way SIMD registers are fully filled only at level 2 and above. Thus, Intra-SIMD will only be effective on trees that can heavily be pruned from these numerous scalar nodes. The **Inter-SIMD** vectorization does not suffer from this problem since SIMD register lanes are filled by LLRs and bits from multiple frames instead. However, the decoder needs to wait for enough frames to arrive —which increases latency— and to interleave [109] the LLRs from these frames (*gather*) before proceeding. It also needs to de-interleave the resulting data bits after decoding (*scatter*).

10.4 Evaluation

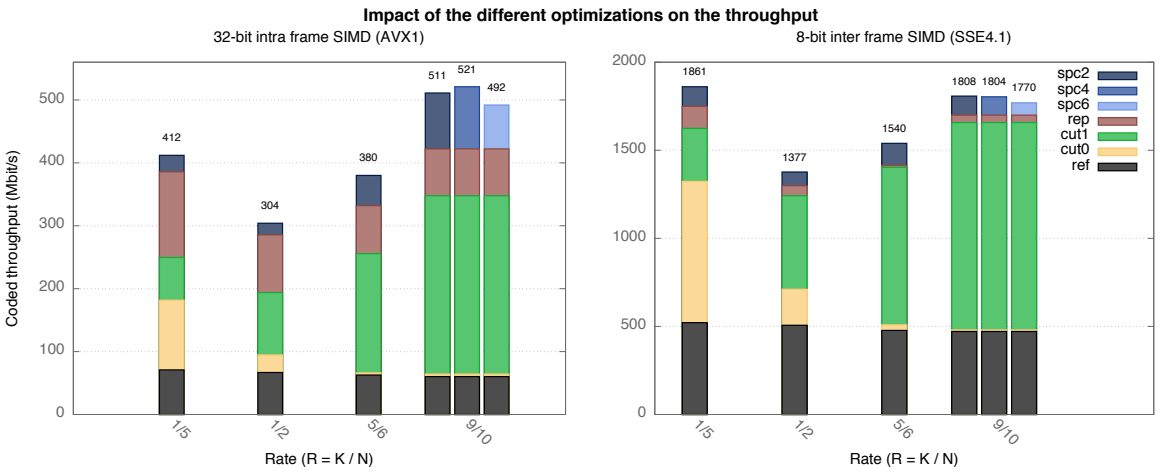


Figure 10.5 – Throughput depending on the different optimizations for $N = 2048$, for intra-frame vectorization on the left and intra-frame vectorization on the right, resp. (on the Intel Xeon CPU E31225).

For concision, we only report the evaluation part on the comparative impact of each

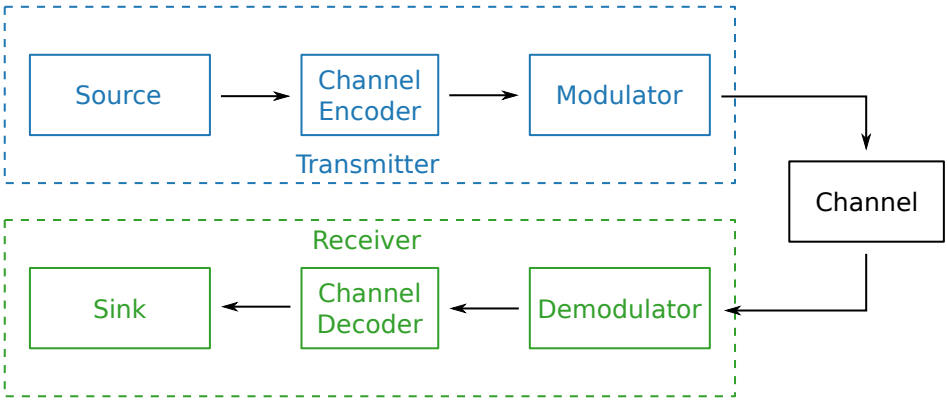


Figure 10.6 – The AFF3CT communication chain.

optimization here, to illustrate the benefit of a flexible experimentation framework. The full evaluation study and comparison to the state of the art can be found in the initial paper [47]. The platform used is an INTEL XEON E31225 3.10GHz, with 256KB L2 and 6M L3 caches. Compiler is GNU g++ 4.8. Each measure is obtained as the best of ten runs of a 10 second simulation, taking into account frame loading and result storing. SNR (Signal Noise Ratio) is set to 2.5 dB for tests with 1/5 and 1/2 rates, and to 4.0 dB for the 5/6, 0.84, and 9/10 rate tests.

The tree pruning step has a dramatical effect in general. For example, the reference code for a rate of 1/2 has 2047 nodes, whereas only 291 nodes remain in the pruned version. However, the individual effect of each rewriting rule is not trivial. The plots in Figure 10.5 show the respective impact of several rewriting rules (cuts, repetitions, single parity checks (SPC)), with $N = 2048$ and multiple code rates, for Intra-SIMD and Inter-SIMD respectively. The purpose of the plots is to show that no single rewriting rule dominates for every code rate, and that the respective impact of each rule may vary a lot from rate to rate, making the case for the flexible, extensible architecture of P-EDGE. Indeed, P-EDGE's rewriting rule set can also be enriched with rules for specific ranges of code rate. For instance, the rule *Single Parity Check (SPC)* has been applied with different level limits for a 9/10 code rate, where it has a significant impact and may benefit from fine tuning.

10.5 Discussion

Building on this first success on the polar codes family, the framework has since evolved into a full featured toolbox for developing, experimenting with, and validating a wide variety of forward error correction codes families, and has been renamed AFF3CT [46] (*A Fast Forward Error Correction Toolbox*) in the process, under the impulse of Adrien Cassagne's PhD Thesis research and engineering work. Additional default stages have been added to the simulation chain (see Figure 10.6), and the toolbox now ships with many alternatives for each stage: source generators, encoders, signal modulators, channels with specific noise characteristics, etc. Moreover, optional stages such as the quantization and the puncturer steps can be enabled on demand as well. Finally, the monitoring modules enable various observation probes on the simulation process, for instance to compute statistics such as the residual bit error rate (BER) and frame error rate (FER) after correction, the two main metrics

of the correcting performance of ECC code algorithms. While the polar code specialization technique described above is largely specific to that code family, the vectorization support based on the MIPP wrapper library [44] and the ability to work on various fixed-point and floating-point fundamental data representations are pervasive within the toolbox, thanks to the genericity integrated at the heart of AFF3CT's design.

Another key aspect of AFF3CT's design is its modularity and its architecture inspired from component-oriented programming models [159]. Indeed, AFF3CT defines optional characteristics that can be implemented by modules to enable building high performance simulation chains. To be compliant with such a high performance chain, a module defines some elementary operations, named *tasks*, each implemented by a method. A task method must define a series of input and/or output *sockets*, that is input and/or output data vectors by which data frames will flow throughout the simulation. A high performance chain is then built by binding the output socket(s) of a module's task to the socket(s) of the task(s) of the next module(s) in the chain. Once a full chain is bound, the simulation can be started and proceed without any management step that could slow it down. Moreover, the component-inspired modular architecture of AFF3CT is compatible with the event-driven SYSTEMC/TLM (Transaction Level Modelling) simulation interface specification [136, 157] used within the community to build and run mixed communication chains; that is, chains made of compliant software and/or hardware modules.

Beyond vectorization, a simulation chain can be sped-up further by using multi-threaded parallelization. The support for parallelization of the chain simulation is in itself straightforward, since a simulation chain is basically a Monte-Carlo process, and thus inherently embarrassingly parallel. Likewise, leveraging distributed-computing through, say MPI, is not much challenging either. However, both techniques are mostly reserved to simulation, and not to run "real" production chains due to their impact on latency. The same limitation applies to the use of accelerators such as GPUs. For real production chains, parallelization has instead to resort to pipelining techniques, since the flow of frames representing the communication signal is naturally sequential.

Overall, the AFF3CT toolbox is now reaching maturity. Among the next challenges, working on a tight integration of FPGA support within the toolbox's chains is likely among the main priorities. FPGAs are already frequently used within the communication community, in particular as a cost-effective prototyping/validation step in the expensive process of producing ASICs, and more generally thanks to their prominent aptitude at specialization. As the processing power of general purpose CPUs increases, and FPGAs are becoming increasingly ubiquitous, the benefits in terms of reduced costs and flexibility of running whole communication chains "in software", with the option to offload some key steps on a FPGA, are gradually outweighing the drawbacks in terms of bandwidth, latency and energy consumption.

Another priority challenge is to make AFF3CT available to a wider community of users. In this respect, the C++ API defined by the AFF3CT library, though very powerful, is likely too low level for scientists and engineers in communication electronics that do not have a deep background in modern C++ computer programming. We thus also have ongoing works on designing high-level language bindings on top of the AFF3CT library, with a specific focus on supporting MATLAB at this, likely the most largely used environment in this domain. Even though designing such a support is mainly a matter of technical engineering, the deep differences in scope, spirit and functionalities between C++ on the one side, and the MATLAB environment with its scripting language on the other side, makes the endeavour rather tricky.

10.6 Conclusion on Tuning through Specialization

In this chapter, we studied an applicative code tuning approach based on specialization through source code generation, based on the work of Adrien Cassagne as part of his PhD Thesis. This work deals with the design of high performance decoders for a family of error correction code algorithms named *polar codes*. The typical implementation layout of this family of ECC decoders makes them difficult to handle by regular optimization tools, such as the vectorizing engines of general purpose compilers. By taking advantage of known simplification rules, the P-EDGE framework was designed to specialize a generic polar decoder implementation for programmer-selected polar code instances by leveraging source code rewriting techniques. We also discussed evolutions and perspectives from this, and especially one that has already been heavily instantiated, namely the extension of the scope of the framework —still as part of the PhD work of Adrien Cassagne— to a full feature software ECC implementation toolbox to build whole ECC communication chains for the prototyping, the simulation, the validation of ECC algorithms for many popular classes, and even their exploitation in production contexts; and as part of this broad scope enlargement, the framework has since been renamed AFF3CT. We also discussed additional work directions, regarding the parallelization of simulation chains, the ability to accommodate hardware components and particularly FPGAs within these chains, as well as enhancing programmability through high level languages, to engage users beyond C++-literate programmers.

This chapter also concludes the third and last part of this manuscript, in which we have examined multiple ways to render applicative codes eligible for HPC, that were initially out of scope due to peculiar characteristics. The code tuning techniques we examined involved task granularity control, code transformations to enable or improve auto-vectorization, and program specialization with respect to select exploitation instances through rewriting-based code generation. Next chapter will now propose a general conclusion and a series of perspectives for this manuscript as a whole.

Conclusion

Chapter 11

Conclusion

This last chapter concludes the manuscript. It first gives an overview of the topics we explored and summarizes the overall outcome of the exploration. Then, in a second part, it discusses some perspectives, in relation with the evolution we are currently witnessing of the High Performance Computing domain.

11.1 Outcome

The purpose of this manuscript was to look back at the experience I gathered over the last twenty years in designing instruments of productivity for the High Performance Computing (HPC) domain, to ponder the lessons learnt, to discuss what remains to be done and what should be explored next. For the sake of clarity, we divided the analysis along three axes, that is, three complementary classes of approaches to enhancing the productivity in developing applicative codes for HPC: the performance portability class here examined from the point of view of runtime systems, the programmability class of means to write and build HPC codes, and the tuning class to adapt and optimize existing codes to HPC.

Performance Portability In the first main Part, page 11, we discussed the increasingly critical role of runtime systems in providing HPC applicative code programmers with an effective performance portability. We explored communication runtime systems from the cases of the MADELEINE and NEWMADELEINE communication libraries, computing runtime systems from the cases of the MARCEL thread library and the STARPU task-based runtime system, as well as the extension of STARPU as a distributed computing runtime system to combine computations and communications. We observed common principles in runtime system designs: an architecture layout separating an abstract, device-independent layer from an extensible driver-based device-dependent layer —to provide the portability—; and an optimizing engine —to provide performance—. In particular, processing user-level work requests through an asynchronous, programmable scheduling engine such as the NEWMADELEINE core or STARPU core results in proactive and versatile runtime systems for communication and for computation respectively. However, such schedulers necessitate an adequate level of expressiveness and semantic directives from the associated programming model to accurately determine the constraints to be enforced and from that, the licence to apply optimization strategies without putting the applicative code correctness at risk. Furthermore, we studied the properties of the Sequential Task Flow (STF) programming model used by STARPU for

both parallel and distributed computing —modeled after the dependence analysis used in modern processors to automatically extract instruction-level parallelism— to let programmers express parallelism easily while preserving the sequential applicative algorithm layout, and to build a systematic virtual ordering of events at the heart of the STF ability at scalability.

Programmability In the second main Part, page 61, we investigated the topic of enhancing programmability further, about approaches and tools to help designing and assembling HPC applications. We studied three approaches: 1) Leveraging a parallel language to easily write parallel applications targeting a parallel computing runtime system; 2) Leveraging composition and enforcing interoperability to assemble HPC software from individual building blocks; 3) Implementing a separation of concerns in HPC kernel development to let the core algorithmic design process and the hardware specific optimization process be enacted by distinct persons. Following the first approach, using the KSTAR OPENMP compiler, we were able to target the STARPU runtime system while keeping the required expertise to a moderate level from the programmer. Moreover, we showed how a compiler such as KSTAR enabled experimenting with OPENMP extensions, and how an adequate extension can make the work of the programmer easier and safer, while substantially enhancing the level of performance. We then studied the building of composite HPC codes, first through the NETIBIS composable communication subsystem for *grid* computing, enabling programmers to dynamically assemble bespoke network stacks on the spot, and second through the interoperability APIs designed as part of the European project H2020 INTERTWINE to let multiple pieces of HPC software —each built on its own runtime system— be assembled together without interference by having their respective runtime systems coordinate their resource allocation and usage. Finally, we studied the INKS model designed to enable separation of concerns between the programmer in charge of developing some domain specific scientific simulation algorithm and the programmer in charge of exercising computer science expertise to select and apply machine specific optimization strategies. Thanks to the INKS model properties and the associated compilers, the *vanilla* algorithm and the optimization strategies can be kept separate in distinct source files, thus enhancing programmability, and performance portability as well.

Tuning In the third and last main Part, page 101, we looked into the case of some programs that do not directly fit the patterns and/or properties expected by general purpose HPC tools, and explored tuning instruments to bring such programs closer from a HPC tools "sweet spot". The first case we studied concerned the matter of codes that generate tasks with a computational weight too low for the individual scheduling process of such tasks to be profitable. We saw how the TAGGRE framework enables the programmer to apply systematic task aggregation schemes as a pre-processing stage, to make the scheduling work worthwhile. The second case concerned a class of applicative codes that could possibly take advantage of the vector processing instruction sets available in modern processors, but are prevented to do so due to issues such as data layouts or data access patterns. We examined a methodology based on the MAQAO assembly analysis framework to detect such issues in compiled binaries to suggest and evaluate source code restructuring hints. Finally, the third case we delved into concerned the optimization of the family of *Polar* Error Correction Codes decoders within the P-EDGE/AFF3CT framework. Due to its natural expression, and especially to internal dependences, the generic algorithm is not directly vectorizable.

However, thanks to a special purpose code generator implementing known rewriting rules, the framework is able to specialize decoders on a per-code basis, and in doing so, to extract vectorizable parallelism.

Assessment In the end, this exploration led us to inquire into many areas of the HPC domain, from communication to computation, from parallel computing to distributed computing, and along the chain of HPC tools from compilers all the way to a binary analyzer, also with a deeper focus on runtime systems, that remain my main research topic. This main research topic on runtime systems led to the study of many key relationships: The relation between work and data, essential for the sequential task flow model, but also at the heart of the INKS model and the code restructuring study with MAQAO; The articulation between programming languages and native runtime library APIs; The relation to establish between the application side and the hardware side to ensure the portability of performance; The relation between multiple HPC pieces of code inside a composite application to implement a coordinated access to hardware resources; And also the relation between theory and practice to let algorithmic theoreticians more easily implement and put their algorithms to use, thanks to the integration of programmable scheduling engines in the architectures of the NEWMADELEINE and STARPU runtime systems. As announced in the Introduction, this exploration did not intend to be exhaustive, and, naturally, many other important HPC topics have not been visited during this exploration, for instance the matter of Input/Output and storage, the matter of cluster-wide job scheduling and placement, or the increasingly critical matters of fault tolerance and of lowering the energy consumption. These two last topics of fault tolerance and energy stand among the perspectives we sketch now.

11.2 Perspectives

Due to the effect of scale that impacts virtually every aspects of HPC, from the ever increasing computing platforms size and intricacy to the increasingly complex HPC applications, the perspectives for the instruments of productivity in HPC are abundant. We try here to delineate some of the major trends and challenges.

Runtime Systems: Harnessing HPC Evolutions

The *Exascale* era, initially announced for a few years ago, is very likely arriving with the next batch of top supercomputers, in the early 2020's years. While the number of applications really drawing a full exaFLOPS of computing power will probably be small, at least in the beginning, one can expect large petaflopical applications to become common, if not mainstream, in the workload of such upcoming supercomputers. Yet, even such petaflopical codes remain a challenge for the state of the art runtime systems, except for embarrassingly parallel applications, thus the perspective of exaflopical codes brings in even more pressure.

On the deployment and networking side, the application startup-time, and especially the connectivity establishment step must be kept at a level low enough not to be prohibitive, which suggests favoring open session models and sparse connectivity. Moreover, the networking runtime system, and likely the distributed computing runtime as well, will have to take the network topology into account. Indeed, early experiments on some modern platforms confirm that performances can dramatically change between jobs assigned to different sets

of nodes, due to different routers and physical connectivity encountered in each node set. A tight cooperation between supercomputers batch schedulers and runtime systems will certainly be helpful to reduce the impact of this issue.

In terms of computing hardware, accelerators of the GPU kind will probably remain the principal HPC workhorses in the foreseeable future. Their raw power advantage on HPC workload remains unchallenged, while many shortcomings of the early GPU samples have now been alleviated, and the addition of features such as half-precision instructions has increased their versatility. Following a somewhat opposite evolution path, reconfigurable devices (FPGA) —featuring built-in versatility— are becoming increasingly powerful while maintaining their advantage on power consumption. However, even though the StarPU runtime system has demonstrated the effectiveness of a dynamic offload decision mechanism for GPU-based heterogeneous nodes, the prohibitive cost of a FPGA device reconfiguration still severely limits the scheduling options for StarPU. Moreover, the case of GPUs is not definitively settled either, as the huge power and level of parallelism of high-end modern GPUs can become prohibitive to be managed entirely from the on-CPU runtime system scheduler. Recent experiments on such devices tend to indicate that some form of embedded on-GPU scheduler would be needed to manage parallelism at a finer grain locally, while the main on-CPU scheduler would concentrate on a coarser grain of parallelism [91, 151].

Numerous perspectives exist on the software side. We can expect an increasing pressure on the interoperability front to enable more HPC code reuse through composition, because the cost and expertise required to develop for HPC make it increasingly unviable for application developers to implement everything but the application specifics "in-house". There is also an urgent need for tools to analyse and debug performances, as the dynamic decisions of runtime systems associated to the application scale can make it challenging to pinpoint the root cause of performance issues. In this respect, approaches based on simulation, such as with the SimGrid framework, are of particular interest due to their ability to reproduce executions in a controlled environment, and also for their ability at fast, inexpensive and accurate experiments with scheduling algorithms on large simulated platforms.

The energy parameter has to increasingly be factored-in in the scheduling engine of computing runtimes. StarPU has integrated for many years already —as part of the work of the European Project FP7 PEPPHER [103]— the possibility to assign energy costs to tasks' kernels and minimize such energy costs while scheduling, however the capability has not been much used in practice so far. Yet, the pressing needs to reduce the energy consumption in computing is likely to increase the interest in this functionality. Moreover, in the perspective of having to dynamically choose between a GPU and a FPGA unit at scheduling time, the energy criterion would be essential to avoid over-selecting the GPU, as FPGAs are more energy efficient than GPUs [132]. This, though, will be highly dependent on the availability of accurate and comprehensive hardware registers for monitoring the energy consumption, not only of the CPU cores, but also of the devices, and importantly also of the memory. We indeed observed that selecting a low CPU energy demanding, but slower, kernel execution option was not necessarily a win in the end, due to the cost of refreshing the memory for a longer time [45].

As discussed in Section 4.3, the integration of automatic distributed load-balancing mechanisms in StarPU stands out among the popular feature requests. Indeed, while the single-node StarPU mode handles most aspects of a task-based HPC code execution even on a heterogeneous, accelerated computer, the distributed StarPU mode instead leaves more details up to the application programmer, especially regarding the data distribution and the

inter-node load balancing. Fulfilling this request is among the ongoing works, regarding the specific situation of re-balancing the workload distribution after a failure recovery, and will be among the priority future works for a more generic load-balancing subsystem.

Fault tolerance is now really becoming a top priority issue for distributed computing runtime systems. Not only the event of a failure is not hypothetical —especially in a large computing session— due to the number of hardware components involved, but the wasted cost and energy resulting from a non-recoverable issue in such a session are increasingly unacceptable from an economical as well as a societal point of view. Here, the main challenge for computing runtime systems will be to use work requests' semantic metadata, such as task/data dependence annotations, to determine the smallest set of requests to replay/recompute in case of failure for the application to recover. Another challenge will be to articulate the various possible levels in fault tolerance, such as, for instance, the applicative code algorithm level versus the runtime system level, to make them cooperate instead of interfering. In any case, such runtimes would need to implement their own failure detection mechanisms, or better, to rely on some standard approach, and in this, the adoption of a "fault tolerant MPI" specification —still being discussed at the time of this writing— would establish a robust, enabling foundation.

Therefore, fault tolerance mechanisms are also naturally a high priority issue in the coming years for communication runtime systems such as NEWMADELEINE, at least to enable an orderly error detection, handling, and reporting, in order to let higher level software layers decide on the actions to take upon failure. The other major upcoming mission for these communication runtimes, as sketched in Section 2.5, will be to take advantage of the increasingly rich offloading capabilities of common network processing on high performance communication devices, while preserving performance portability, here again with the joint purposes of scalability and energy efficiency.

Programming Models and Tools: Addressing the Diversity of Programmers, the Diversity of Applications

The common trend on the theme of models and tools design for programming is to address the diversity: the diversity of programmers and also the diversity of applications.

Addressing the diversity of programmers is a necessity as HPC becomes widely available and open to a large number of users. Initiatives such as ETP4HPC and PRACE-6IP in Europe devote an important effort to bring more users to HPC, especially from small and medium scale enterprises/industries. National-scale supercomputers such as those from GENCI in France, as well as regional supercomputing centers are open to both academic researchers and industries. Also, the quick development of "supercomputers as a service" —even equipped with GPU accelerators— from cloud computing suppliers make HPC resources readily available to virtually anyone with a few mouse clicks and a credit card. The numerous PhD Theses in the literature each devoted to the port and optimization of a single application on some HPC platform gives an idea of the challenge to convert this readily available computing power into successfully and efficiently obtained application results.

Annotation-based languages should therefore remain popular in the coming years, thanks to their ability to absorb a significant part of the complexity in HPC programming with moderate changes to existing codes. Among them, OPENMP should remain particularly favoured thanks also to its widespread availability in every modern compilers and to its versatility. Yet, as we discussed in Chapter 5, and despite the shear size of the latest OPENMP specification

revisions, there is room to widen its scope, especially on the three following topics: Making *data* a first class citizen of OPENMP, on par with *work*, to let both the compiler and the targeted runtime system have a better understanding of the applicative code pattern, and therefore be able to do more on behalf of the programmer, for instance more automatically targeting, and scheduling work on, accelerated nodes —some effort is already ongoing in that direction with the support for deep copies from/to accelerator devices—; Handling distributed computing, or at least facilitating the interaction of OPENMP with some distributed computing runtime, to avoid programmers reinventing the wheel, with varying degrees of success, in countless OPENMP + MPI applications —here also, some research effort is underway, for instance with BSC's task-aware MPI support, but this still requires some expertise from the application programmer—; Making OPENMP more aware and more cooperative with its surrounding ecosystem through interoperability, which would for instance enable to use OPENMP both in an application and in a library linked with that application without issues.

C++ approaches such as KOKKOS [70] and RAJA [27], as well as SYCL [155] are on a rising trajectory, in parallel somehow as the evolution of the C++ language itself. Such frameworks position themselves rather more on the expert side than annotation based languages. As a counterpart, their homogeneous integration in C++ enables compilers to optimize applications using them in a global and comprehensive way, while there is some unavoidable "discontinuity" and thus some loss of information between the OPENMP statements lowering stage and the native language lowering stage in OPENMP compilers. Moreover, OPENMP-like single code CPU+accelerator programming is also possible with C++ using dedicated compilers such as CODEPLAY's COMPUTECPP compiler for SYCL, or INTEL's Data Parallel C++ compiler for the ONEAPI extension of SYCL. Yet, even though single code hybrid programming may be desirable from a code management point of view, it also goes in a direction opposite to some no less desirable objectives such as the separation of concerns and the interoperability with the HPC ecosystem. Reconciling these opposite aspects will likely be among the main challenges for this topic.

Skeleton-based programming models such as SKEPU 2 [74] aim at conciling several aspects: Taking advantage of modern C++ capabilities, while offering user friendlier constructs than, for instance, SYCL, for a series of parallel computing patterns commonly found in HPC applications such as *stencils* and *map/reduce* patterns. They are therefore essential in the process of bringing more programmers to HPC, by avoiding them to reimplement and optimise these popular patterns. Moreover, a skeleton framework such as SKEPU 2 not only handles parallelizing the code on a CPU via OPENMP or task-based runtime systems, by also handles the burden of targeting accelerators. Some of their upcoming challenges will be to handle more complex, hierarchical patterns, such as for instance adaptive mesh refinement (AMR) patterns, handling more complex backends combinations such generating parallel tasks, nesting for example a parallel OPENMP loop within a STARPU task. Skeleton frameworks could also bring new opportunities on addressing the open question of task granularity choice, by taking advantage of the application pattern's knowledge.

Another strong trend toward an "inclusive" parallel programming approach is about offering APIs for languages such as JULIA or PYTHON, since these languages have a much more gentle learning curve than C++ approaches or annotated C approaches, and are able to hide most technical HPC-related details from application programmers. Some effort is indeed currently underway to target STARPU from a JULIA program.

Addressing the diversity of applications becomes a priority also, as on the one side, workloads have themselves diversified, with notably the "Big Data" processing class and

the Machine Learning class of applications that have now overpassed the HPC class; and on the other side, some of the techniques developed for HPC are useful for these other kinds of workloads, driving the convergence we are now witnessing between HPC, Big Data and Machine Learning [122].

One example is the case of frameworks used in the Big Data domain, such as APACHE SPARK mentioned in Section 6.1.3, which offer programming means not entirely dissimilar from those of the skeleton frameworks, though with slightly more abstraction and with a lot more software layers than C++ skeletons. SPARK offers interfaces in the SCALA and JAVA languages, both targeting the JAVA virtual machine (JVM), as well a PYTHON interface. As discussed previously, it lets programmers express parallelism on partitioned and distributed collections of objects, processed through map/reduce like fundamental methods. A working prototype JAVA API has been developed on top of STARPU, implementing the SPARK programming model look and feel, with the objective of widening the scope of workloads beyond HPC applications. Due to the specificities of the SPARK model, the individual elements of SPARK's collections objects are also objects by design, even when native types such as integers, floats or double elements would be sufficient, which incurs significant overhead for computing and for communicating. On the computing side, the challenge will therefore be to devise how to transition to native types when possible, to speed up processing, while preserving the programming model elegance and simplicity. And, on the communication side, the objective will be to optimize transfers, either through native types messaging when possible, or by leveraging, for instance, the fast serialization techniques developed within NETIBIS and the IBIS project as a whole. A third, cross-cutting challenge will be to address the differences in HPC and Big Data system designs, usages and cultures: As summarized in the US NITRD 2019 report on these domains convergence [122], "*At the system level, there is a significant gap between how HPC systems are designed (tightly coupled collections of homogeneous nodes) versus BD* [Big Data] *systems (based on cloud computing data center architectures that consist of large numbers of loosely coupled and possibly heterogeneous computing nodes). These structural differences, in turn, have led to a split in the software stack that is both technological and cultural. The HPC stack relies on tools developed in government laboratories and academia. In contrast, the BD stack is much larger and more varied and is often driven by open-source projects, with the main contributors being commercial entities.*"

Regarding the case of the Machine Learning domain, HPC techniques would indeed be useful also, and in fact, they are already largely employed. For instance, the popular TENSORFLOW framework from GOOGLE is built on a dataflow model, with support for heterogeneous, accelerated and distributed computing. Thus, while Machine Learning oriented frameworks and HPC oriented frameworks will likely benefit from each others' improvements, and perhaps interbreed at some point, the Machine Learning domain will probably not be a major source of entirely specific requirements on HPC instruments of productivity. Significant differences between HPC and Machine Learning are rather observed at the level of supercomputer usages, where typical Machine Learning jobs tend to be more agile and thus less predictable than HPC jobs, leading supercomputing providers such as the french GENCI agency to offer distinct partitions and different resource request modalities for each.

Adaptation and Transformation Tooling: Making Decisions, Helping with Transitions

Finally, a probable important trend in productivity instruments for HPC in the coming years will be the design of tools and methodologies for adapting or transforming programs into HPC-enabled software and improving existing HPC applications, along the line of the works presented in Part III. Due to their context of utilization, many of these approaches will be specific to their target case, such as the code generator used for polar decoders in the P-EDGE/AFF3CT tool (see Chapter 10). However, the case of transitioning to and adapting task parallelism that we discuss now should have a reasonably broad scope.

We have studied the benefits of task-based parallel programming models in Chapter 3 and Chapter 4. Designing an application from scratch along a task-oriented paradigm can be relatively straightforward, either directly or through means mentioned above such as skeleton frameworks. In contrast, transforming a pre-existing application to adopt a task-oriented paradigm is much more demanding, when restarting from scratch is not an option. Task extraction tools would therefore be helpful to guide programmers in the process of some programming model transitioning towards task parallelism. Past works have demonstrated the feasibility of the principle, such as the ASTEX [128] tool at the basis of the *Codelet finder* tool from Company CAPS ENTREPRISE in the mid-2000 years, or the CERE tool [51] (*Codelet Extractor and REplayer*). Building on these works, one of the missing part to fill would be the ability to associate some computational weight estimation to the extracted codelet. In this, the capabilities of the MAQAO tool (see Chapter 9) could provide the means to generate such an estimation.

Closely related to the issue of task extraction is indeed the question of deciding about the granularity of tasks. The choice of a task granularity, as discussed in Section 3.5.2, is indeed a matter profitability, and a matter of resolving the trade-off of enabling sufficient load balancing while keeping the scheduling overhead under control. While the TAGGRE tool presented earlier (in Chapter 8) offers a straightforward solution for a specific case, the general case is difficult, as the solution depends on many factors such as the runtime system used, the scheduling algorithm selected, the number and heterogeneity of computing resources, the other tasks, the application algorithm layout and degree of parallelism, and also other aspects such as, for instance, multiple implementations available for a given task kernel. In practice, current approaches more or less rely on a combination of rules of thumb, scheduling overhead estimation through micro-benchmarks, and trial and error. Again, leveraging MAQAO, or perhaps building a computational weight estimation of kernels on the compiler side could provide some more useful hints.

An alternative, explored as part of the ongoing PhD thesis of Gwénolé Lucas, is to define special tasks called bubbles, with multiple "views" of different granularity, and defer the selection of a view at run-time, to take into account the computing resources current occupancy in the view selection decision. The decision will also have to take into account other factors such as for instance the performance estimation of each view on available resources, and again whether some views would have a computational weight less profitable than others on these resources.

More generally, the process of dynamic decision making, for a runtime such as STARPU, could see its scope widen in the future, to encompass not only the scheduling step, the computing device selection or some kernel's view selection just mentioned, but also deciding about triggering local, diffusive load re-balancing with neighbours, triggering some more

intrusive global re-balancing steps, triggering some FPGA reprogramming step to anticipate the increased abundance of some kernel in the task-flow lookahead. All these decisions will necessitate tools to provide estimations of the costs and benefits of possible strategies.

All-in-all, even though the two decades covered by this manuscript have witnessed an extraordinary rich and quickly evolving HPC landscape, the coming times should be no less fascinating and busy...

List of Acronyms

ABI Application Binary Interface
AFF3CT A Fast Forward Error Correction Toolbox
AMR Adaptive Mesh Refinement
ANR Agence nationale pour la recherche
AOP Aspect-Oriented Programming
API Application Programming Interface
ASIC Application Specific Integrated Circuit
AST Abstract Syntax Tree
BSC Barcelona Supercomputing Center
CAF Coarray Fortran
CCI Common Communication Interface
CEA Commissariat à l'Énergie Atomique
CERE Codelet Extractor and REplayer
CMT Coarse Multi-Threading
CPU Central Processing Unit
CUDA Compute Unified Data Architecture
DAG Directed Acyclic Graph
DMA Direct Memory Access
DPL Dependent Partitioning Language
DRAM Dynamic Random Access Memory
DSA Dynamic Single Assignment
DSL Domain Specific Language
DSM Distributed Shared-Memory
DSEL Domain Specific Embedded Language
DTD Dynamic Task Discovery
ECC Error Correction Code
ECP Exascale Computing Project
EDR Eighteen Data Rate
FMM Fast Multipole Method

FMT Fine-grained Multi-Threading

ETP4HPC European Technology Platform for HPC

FPGA Field-Programmable Gate Array

GENCI Grand Équipement de Calcul Intensif

GPU Graphic Processing Unit

HAL Hardware Adaptation Layer

HBM High Bandwidth Memory

HPC High Performance Computing

HPF High Performance Fortran

IB Infiniband

ILP Instruction-Level Parallelism

INTERTWinE Programming Model INTERoperability ToWards Exascale

IoT Internet-of-Things

IPL IBIS Portability Layer

JIT Just-In-Time [compiler]

JVM JAVA Virtual Machine

KNL Knight Landing

MCDRAM Multi-Channel DRAM

MMU Memory Management Unit

MSI Modified/Shared/Invalid cache policy

MTBF Mean Time Between Failures

MTL MAQAO Trace Library

MX MYRINET eXpress

NIC Network Interface Card

NIO JAVA New Input/Output

NITRD The Networking and Information Technology Research and Development Program (USA)

NLR Nested Loop Recognition algorithm

NUMA Non-Uniform Memory Access

OCR The Open Community Runtime

OFA OpenFabrics Alliance

ORNL Oak Ridge National Laboratory

SORS STARPU OPENMP Runtime Support

P-EDGE Polar ECC Decoder Generation Environment

PIO Programmed Input/Output

PM^2 Parallel Multi-threaded Machine

PMIx Process Management Interface - Exascale

PTG Parameterized Task Graph
QCD Quantum Chromodynamics
RDMA Remote Direct Memory Access
RPC Remote Procedure Call
RDD Resilient Distributed Datasets
SIMD Single Instruction Multiple Data
SMT Simultaneous Multi-Threading
SP Streaming Processors
SRA Strategic Research Agenda
SRAM Static Random Access Memory
STF Sequential Task Flow
STL Standard Template Library
TLM Transaction Level Modelling
TSVC Test Suite for Vectorizing Compilers
XMP XcalableMP Programming Language

Bibliography

[1] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, 2010.

[2] Emmanuel Agullo, Olivier Aumage, Berenger Bramasi, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, page 14, April 2017.

[3] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[4] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for multicore architectures. *SIAM Journal on Scientific Computing*, 2014.

[5] Muhammed Abdullah Al Ahad, Christian Simmendinger, Roman Iakymchuk, Erwin Laure, and Stefano Markidis. Efficient algorithms for collective operations with notified communication in shared windows. In *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI*, 2018.

[6] A. Alamdar-Yazdi and F.R. Kschischang. A simplified successive-cancellation decoder for polar codes. *Communications Letters*, 2011.

[7] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for industrial and Applied Mathematics, 1992.

[8] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 1995.

[9] Gabriel Antoniu, Luc Bougé, and Raymond Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. In *11 IPPS/SPDP'99 Workshops*, 1999.

[10] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, 2009.

[11] Scott Atchley, David Dillow, Galen Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. The common communication interface (CCI). In *Symposium on High Performance Interconnects (HOTI)*, 2011.

[12] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université de Bordeaux, 2011.

[13] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2011.

[14] Olivier Aumage, Gabriel Antoniu, Luc Bougé, Vincent Danjean, and Raymond Namyst. *Getting started with PM2*. LIP, ENS-Lyon, 2001.

[15] Olivier Aumage, Denis Barthou, Christopher Haine, and Tamara Meunier. Detecting simdization opportunities through static/dynamic dependence analysis. In *Workshop on Productivity and Performance (PROPER)*, 2013.

[16] Olivier Aumage and Luc Bougé. A portable and adaptative multi-protocol communication library for multithreaded runtime systems. In *Workshop on Runtime Systems for Parallel Programming, held in conjunction with the Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

[17] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In *Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*, 2007.

[18] Olivier Aumage, Christopher Haine, and Denis Barthou. Rewriting System for Profile-Guided Data Layout Transformations on Binaries. In *Euro-Par 2017 - European Conference on Parallel Processing*, Santiago de Compostela, Spain, August 2017.

[19] Olivier Aumage, Rutger Hofman, and Henri Bal. NetIbis: An Efficient and Dynamic Communication System for Heterogeneous Grids. In *Cluster Computing and Grid 2005*, 2005.

[20] Olivier Aumage, Guillaume Mercier, and Raymond Namyst. MPICH/Madeleine: A True Multi-Protocol MPI for High Performance Networks. In *15th International Parallel and Distributed Processing Symposium*, 2001.

[21] Guillaume Aupy, Anne Benoît, Brice Goglin, Loïc Pottier, and Yves Robert. Co-scheduling hpc workloads on cache-partitioned CMP platforms. *International Journal of High Performance Computing Applications*, 2019.

[22] Denis Barthou, Gilbert Grosdidier, Michael Kruse, Olivier Pène, and Claude Tadonki. QIRAL: A High Level Language for Lattice QCD Code Generation. In *ETAPS*, 2012.

[23] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cédric Valensi. Performance tuning of x86 openmp codes with maqao. In *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2010.

[24] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, 2001.

[25] Gabriel Bathie, Loris Marchal, Yves Robert, and Samuel Thibault. Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms. Research report, Inria, 2020.

[26] Michael Bauer and Michael Garland. Legate numpy: Accelerated and distributed array computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[27] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. Performance portable c++ programming with raja. In *24th Symposium on Principles and Practice of Parallel Programming*, 2019.

[28] A. J. Bernstein. Analysis of programs for parallel processing. *Transactions on Electronic Computers*, 1966.

[29] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Software, Environments and Tools. Society for Industrial and Applied Mathematics, 1997.

[30] Filip Blagojević, Paul Hargrove, Costin Iancu, and Katherine Yelick. Hybrid pgas runtime support for multicore nodes. In *Conference on Partitioned Global Address Space Programming Model*, 2010.

[31] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Symposium on Principles and Practice of Parallel Programming*, 1995.

[32] ETP4HPC Steering board and SRA Experts. Strategic research agenda (SRA) 2017. Technical report, European technology platform for HPC (ETP4HPC), 2017.

[33] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomaus Hérault, Pierre Lemarinier, and Oleg Lodygensky. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *SC'02 conference on Supercomputing*, 2002.

[34] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczezek, Asim YarKhan, and Jack Dongarra. Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. Technical report, ICL, UTK, 2010.

[35] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 2012.

[36] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Thomas Herault, Piotr Luszczek, and Jack Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. *Scalable Computing and Communications: Theory and Practice*, 2012.

[37] Franccois Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-André Wacrenier, and Raymond Namyst. Structuring the execution of openmp applications for multicore architectures. In *International symposium on parallel and distributed processing (IPDPS)*, 2010.

[38] François Broquedis, François Diakhate, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *International Workshop on OpenMP*, 2008.

[39] Elisabeth Brunet. *Une approche dynamique pour l'optimisation des communications concurrentes sur réseaux hautes performance*. PhD thesis, Université Bordeaux 1, 2008.

[40] Javier Bueno, J. Planas, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Productive programming of gpu clusters with ompss. In *Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[41] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Conference on Supercomputing*, 1988.

[42] João Cardoso, Tiago Carvalho, José Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. LARA: an aspect-oriented programming language for embedded systems. In *11th annual international conference on Aspect-oriented Software Development*, 2012.

[43] Adrien Cassagne. *Optimization and Parallelization Methods for the Software-defined Radio*. PhD thesis, Université de Bordeaux, 2020.

[44] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. MIPP: a portable C++ SIMD wrapper and its use for error correction coding in 5g standard. In *5th international workshop on programming models for SIMD/vector processing (WPMVP)*, 2018.

[45] Adrien Cassagne, Olivier Aumage, Camille Leroux, Denis Barthou, and Bertrand Le Gal. Energy Consumption Analysis of Software Polar Decoders on Low Power Processors. In *The 24nd European Signal Processing Conference (EUSIPCO 2016)*, 2016.

[46] Adrien Cassagne, Olivier Hartmann, Mathieu Leonardon, Kun He, Camille Leroux, Romain Tajan, Olivier Aumage, Denis Barthou, Thibaud Tonnellier, Vincent Pignoly, Bertrand Le Gal, and Christophe Jego. AFF3CT: A Fast Forward Error Correction Toolbox! *SoftwareX*, 2019.

[47] Adrien Cassagne, Bertrand Le Gal, Camille Leroux, Olivier Aumage, and Denis Barthou. An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes. In *The 28th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2015)*, Raleigh, United States, September 2015.

[48] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. PMIx: Process management for exascale environments. In *24th European MPI Users' Group Meeting*, 2017.

[49] Adrián Castelló. *Programming Models for the Lightweight Threads Era*. PhD thesis, Universitat Jaume I, 2018.

[50] Adrián Castelló, Antonio J. Peña, Sangmin Seo, Rafael Mayo, Pavan Balaji, and Enrique S. Quintana-Ortí. A review of lightweight thread approaches for high performance computing. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[51] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization*, 2015.

[52] ThunderX2(r) CN99XX product brief, 2018.

[53] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *International Parallel and Distributed Processing Symposium*, 2011.

[54] Terry Cojean. *Programmation des architectures hétérogènes à l'aide de tâches divisibles ou modulables*. PhD thesis, Université de Bordeaux, 2018.

[55] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Computing*, 2018.

[56] Michel Cosnard and Emmanuel Jeannot. Automatic parallelization techniques based on compact dag extraction and symbolic scheduling. *Parallel Processing Letters*, 2001.

[57] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum. In *Symposium on Operating Systems Principles SOSP'89*, 1989.

[58] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *Conference on Computing Frontiers*, 2008.

[59] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 1991.

[60] Anthony Danalis, George Bosilca, Aurélien Bouteiller, Thomas Hérault, and Jack Dongarra. PTG: An abstraction for unhindered parallelism. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing WOLFHPC*, 2014.

[61] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor. In *International Conference on Computational Science ICCS 2013*, 2013.

[62] Alexandre Denis, Olivier Aumage, Rutger Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC'13)*, 2004.

[63] Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, and Samuel Thibault. Using Dynamic Broadcasts to improve Task-Based Runtime Performances. In *Euro-Par International European Conference on Parallel and Distributed Computing*, 2020.

[64] Saïd Derradji, Thibault Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter. The BXI interconnect architecture. In *23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, 2015.

[65] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, 2013.

[66] James Dinan and Mario Flajslik. Contexts: A mechanism for high throughput communication in openshmem. In *8th International Conference on Partitioned Global Address Space Programming Models*, 2014.

[67] Jirí Dokulil. Consistency model for runtime objects in the open community runtime. *The Journal of Supercomputing*, 2019.

[68] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 1979.

[69] Anastasios Doumoulakis, Ronan Keryell, and Kenneth O'Brien. SYCL C++ and OpenCL Interoperability Experimentation with TriSYCL. In *5th International Workshop on OpenCL*, 2017.

[70] H. Carter Edwards and Daniel Sunderland. Kokkos array performance-portable manycore programming model. In *International Workshop on Programming Models and Applications for Multicores and Manycores*, 2012.

[71] Ksander Ejjaaouani. *Conception du modèle de programmation INKS pour la séparation des préoccupations algorithmiques et d'optimisation dans les codes de simulation numérique ; application à la résolution du système Vlasov/Poisson 6D*. PhD thesis, Université de Strasbourg, 2019.

[72] Ksander Ejjaaouani, Olivier Aumage, Julien Bigot, Michel Mehrenberger, Hitoshi Murai, Masahiro Nakao, and Mitsuhisa Sato. InKS, a programming model to decouple performance from algorithm in hpc codes. In *Euro-Par 2018: Parallel Processing Workshops*, 2018.

[73] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Fourth International Workshop on High-Level Parallel Programming and Applications*, 2010.

[74] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 2018.

[75] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *EuroPVM/MPI*, 2000.

[76] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 1991.

[77] Paul Feautrier and Christian Lengauer. *Polyhedron Model*, pages 1581–1592. Springer US, 2011.

[78] Ian Foster and Carl Kesselman. *Computational Grids*, pages 15—51. Morgan Kaufmann Publishers Inc., 1998.

[79] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *International conference on acoustics, speech and signal processing*, 1998.

[80] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Conference on Programming Language Design and Implementation*, 1998.

[81] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*, 2014.

[82] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. Locality-aware work stealing on multi-cpu and multi-gpu architectures. In *Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.

[83] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. On the scalability, performance isolation and device driver transparency of the ihk/mckernel hybrid lightweight kernel. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[84] P. Giard, G. Sarkis, C. Thibeault, and W.J. Gross. Fast software polar decoders. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[85] Brice Goglin. Managing the topology of heterogeneous cluster nodes with Hardware Locality. In *International Conference on High Performance Computing and Simulation (HPCS 2014)*, 2014.

[86] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In *ISC High Performance Computing*, 2019.

[87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 1987.

[88] Christopher Haine. *Kernel optimization by layout restructuring*. PhD thesis, Université de Bordeaux, 2017.

[89] Christopher Haine, Olivier Aumage, Enguerrand Petit, and Denis Barthou. Exploring and evaluating array layout restructuring for SIMDization. In *27th International Workshop on Languages and Compilers for Parallel Computing*, 2014.

[90] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics. Conference Series*, 2006.

[91] Thomas Herault, Yves Robert, George Bosilca, and Jack Dongarra. Generic matrix multiplication for multi-gpu accelerated distributed-memory platforms over parsec. In *Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2019.

[92] Michael A. Heroux, Jonathan Carter, Rajeev Thakur, Jeffrey Vetter, Lois Curfman McInnes, James Ahrens, and J. Robert Neely. ECP software technology capability assessment report. Technical report, Exascale Computing Project, 2018.

[93] Nathan Hjelm, Howard Pritchard, Samuel K. Gutiérrez, Daniel J. Holmes, Ralph Castain, and Anthony Skjellum. Mpi sessions: Evaluation of an implementation in open mpi. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[94] Daniel Holmes, Kathryn Mohror, Ryan E. Grant, Anthony Skjellum, Martin Schulz, Wesley Bland, and Jeffrey M. Squyres. MPI sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale. In *23rd European MPI Users' Group Meeting*, 2016.

[95] Reazul Hoque, Thomas Hérault, George Bosilca, and Jack Dongarra. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In *ScalA '17: The 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2017.

[96] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference*, 2010.

[97] Andra Hugo. *Composability of parallel codes on heterogeneous architectures*. PhD thesis, Université de Bordeaux, 2014.

[98] Myricom Inc. Myrinet EXpress (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet, 2003.

[99] INTERTWinE Project, 2018.

[100] Richard LaRowe Jr., James T. Wilkes, and Carla S. Ellis. Exploiting operating system support for dynamic page placement on a numa shared memory multiprocessor. In *Symposium on Principles and practice of parallel programming PPoPP*, 1991.

[101] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In *8th international conference on partitioned global address space programming models PGAS'14*, 2014.

[102] Rainer Keller, Steffen Brinkmann, José Gracia, and Christoph Niethammer. Temanejo: Debugging of thread-based task-parallel programs in starss. In *Tools for High Performance Computing 2011*, 2012.

[103] Christoph W. Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff, and Sabri Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE*, 2012.

[104] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *International Conference on Code Generation and Optimization*, 2008.

[105] Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. Challenges and opportunities: From near-memory computing to in-memory computing. In *International Symposium on Physical Design*, 2017.

[106] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 1996.

[107] Maha Kooli, Henri-Pierre Charles, Clément Touzet, Bastien Giraud, and Jean-Philippe Noel. Smart instruction codes for in-memory computing architectures compatible with standard SRAM interfaces. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018.

[108] Benoit Lange and Pierre Fortin. Parallel dual tree traversal on multi-core and many-core architectures for astrophysical n-body simulations. In *Euro-Par*, 2014.

[109] B. Le Gal, C. Leroux, and C. Jego. Multi-gb/s software decoding of polar codes. *Transactions on Signal Processing*, 2015.

[110] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *International Conference on Parallel Processing Workshops*, 2010.

[111] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. A constraint-based approach to automatic data partitioning for distributed memory execution. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[112] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 1996.

[113] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, 2005.

[114] Zoltan Majo and Thomas R. Gross. A library for portable and composable data locality optimizations for numa systems. In *Symposium on Principles and Practice of Parallel Programming, PPoPP*, 2015.

[115] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[116] Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi, and Philippe Olivier Alexandre Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Florianopolis, Brazil, 2015.

[117] Timothy G. Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlić, Sanjay Chatterjee, Josh Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, Benoît Meister, Brian Nickerson, Nick Pepperling, Bala Seshasayee, Sagnak Tasirlar, Justin Teller, and Nick Vrvilo. The open community runtime: A runtime system for extreme scale computing. In *High Performance Extreme Computing Conference*, 2016.

[118] MPI Forum. MPI: A message-passing interface standard version 3.1, 2015.

[119] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. Xcalablemp implementation and performance of nas parallel benchmarks. In *Conference on Partitioned Global Address Space Programming Model*, 2010.

[120] Raymond Namyst. *Pm2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Université de Lille 1, 1997.

[121] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine; A computing environment for distributed architectures. In *Fifth International Conference on Parallel Computing*, 1995.

[122] The convergence of high performance computing, big data, and machine learning. Technical report, The Networking and Information Technology Research and Development Program (NITRD), 2019.

[123] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 1998.

[124] oneAPI, 2019.

[125] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

[126] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, 2020.

[127] David Patterson and John Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 4th edition, 2009.

[128] Eric Petit, Franccois Bodin, Guillaume Papaure, and Florence Dru. ASTEX: a hot path based thread extractor for distributed memory system on a chip. In *HiPEAC Industrial Workshop*, 2006.

[129] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high performance CPU programming. In *Conference InPar*, 2012.

[130] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *International symposium on microarchitecture MICRO*, 2013.

[131] Gregory W. Price and David K. Lowenthal. A comparative analysis of fine-grain threads packages. *Journal of Parallel and Distributed Computing*, 2003.

[132] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, , and Phillip H. Jones. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. *Electrical and Computer Engineering Publications. 218.*, 2019.

[133] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R. de Supinski, Sally A. McKee, Petar Radojković, and Eduard Ayguadé. Another trip to the wall: How much will stacked DRAM benefit hpc? In *International Symposium on Memory Systems MEMSYS '15*, 2015.

[134] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 1993.

[135] CORPORATE Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 1993.

[136] Adam Rose, Stuart Swan, John Pierce, and Jean michel Fern. Transaction level modeling in SystemC. Technical report, Cadence Design System, 2004.

[137] Corentin Rossignon. *Un modèle de programmation à grain fin pour la parallélisation de solveurs linéaires creux*. PhD thesis, Université de Bordeaux, 2015.

[138] Corentin Rossignon, Pascal Hénon, Olivier Aumage, and Samuel Thibault. A NUMA-aware fine grain parallelization framework for multi-core architecture. In *PDSEC - International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2013.

[139] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS, 1996.

[140] Emmanuelle Saillard. *Static/Dynamic Analyses for Validation and Improvements of Multi-Model HPC Applications*. PhD thesis, Université de Bordeaux, 2015.

[141] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. PARCOACH: Combining static and dynamic validation of MPI collective communications. *International Journal of High Performance Computing Applications*, 2014.

[142] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking mpi primitives with task-based programming models. *Parallel Computing*, 2019.

[143] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W.J. Gross. Fast polar decoders: Algorithm and implementation. *Journal on Selected Areas in Communications*, 2014.

[144] Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and LMSR: algorithms for dynamic repartitioning of adaptive meshes. *Transactions on Parallel and Distributed Systems*, 2001.

[145] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. Argobots: A lightweight low-level threading and tasking framework. *Transactions on Parallel and Distributed Systems*, 2018.

[146] Marc Sergent. *Passage à l'echelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense*. PhD thesis, Université de Bordeaux, 2016.

[147] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System. In *21st International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2016.

[148] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 1948.

[149] Elliott Slaughter. *REGENT: A high-productivity programming language for implicit parallelism with logical regions*. PhD thesis, University of Tennessee, 2012.

[150] Society of Petroleum Engineers. SPE comparative solution project, 2001.

[151] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic scheduling on gpus. *ACM Transactions on Graphics*, 2012.

[152] Thomas L. Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. BEOWULF: a parallel workstation for scientific computation. In *International conference on parallel processing*, 1955.

[153] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. LIFT: A functional data-parallel IR for high-performance gpu code generation. In *International Symposium on Code Generation and Optimization (CGO)*, 2017.

[154] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[155] Khronos OpenCL Working Group SYCL subgroup. *SYCL Provisional Specification*. Khronos, 2014.

[156] SUMMIT at ORNL leadership computing facility, 2018.

[157] IEEE standard SystemC(R) language reference manual, 2006.

[158] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1st edition, 1998.

[159] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2nd edition, 2011.

[160] Yuan Tang, Rezaul Alam Chowdhury, Chi-Keung Luk, and Charles E. Leiserson. Coding stencil computations using the pochoir stencil-specification language. In *HotPar*, 2011.

[161] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in SkeTo skeleton library. In *15th International Euro-Par Conference on Parallel Processing*, 2009.

[162] Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, Gheorghe Almasi, and Jesus Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 2012.

[163] Xavier Teruel, Vicen c Beltran, Olivier Aumage, Kevin Sala, and Marc Marí. D4.4 Report on runtime systems implementation of resource management API. Project deliverable, H2020 INTERTWinE, 2017.

[164] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications*. PhD thesis, Université Bordeaux 1, 2007.

[165] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar*, 2007.

[166] Martin Tillenius. *Scientific Computing on Multicore Architectures*. PhD thesis, Uppsala Universiteit, 2014.

[167] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Transactions on Parallel Distributed Systems*, 2002.

[168] François Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst. Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager. In *EuroPVM/MPI*, 2007.

[169] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: An efficient java-based grid programming environment. In *Joint ACM-ISCOPE Conference on Java Grande*, 2002.

[170] Brice Videau, Vania Marangozova-Martin, and Johan Cronsioe. BOAST: bringing optimization through automatic source-to-source transformations. In *International symposium on embedded multicore/manycore system-on-chip (MCSoC)*, 2013.

[171] Brice Videau, Vania Marangozova-Martin, Luigi Genovese, and Thierry Deutsch. Optimizing 3d convolutions for wavelet transforms on cpus with sse units and gpus. In *International europar conference on parallel processing*, 2013.

[172] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *International Symposium on Parallel and Distributed Processing*, 2008.

[173] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architure News*, 1995.

[174] Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, 2012.

[175] Afshin Zafari, Martin Tillenius, and Elisabeth Larsson. Programming models based on data versioning for dependency-aware task-based parallelisation. In *International Conference on Computational Science and Engineering*, 2012.

[176] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: A PGAS Extension for C++. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

-=-=-=-