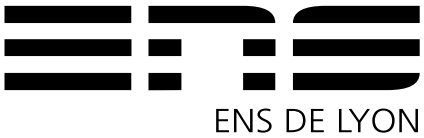
Efficient Use of Task-based Parallelism in HPC Parallel Applications

Romain Pereira



THESE

en vue de l'obtention du grade de Docteur, délivré par

l'ÉCOLE NORMALE SUPÉRIEURE DE LYON

École Doctorale N° 512

INFOMATHS Informatique Mathématiques

Discipline: Informatique

Soutenue publiquement le 27/11/2023, par:

Romain PEREIRA

Exploitation Efficace de l'Asynchronisme de Tâche pour les Applications Parallèles HPC

Devant le jury composé de:

COTI, Camille	Professeure	ETS Montréal	Rapporteure
NAMYST, Raymond	Professeur	Université de Bordeaux	Rapporteur
BRUNET, Elisabeth	Maître de conférences	Telecom SudParis	Examinatrice
CÉRIN, Christophe	Professeur	Université de Paris XIII	Examinateur
DESPREZ, Frédéric	Directeur de Recherche	Inria	Examinateur
CARRIBAULT, Patrick	Ingénieur Chercheur	CEA	Examinateur
ROUSSEL, Adrien	Ingénieur Chercheur	CEA	Examinateur
GAUTIER, Thierry	Chargé de Recherche - HDR	Inria	Directeur de thèse

Résumé

La simulation numérique est un outil puissant pour la recherche scientifique et l'industrie, il est considéré comme le troisième pilier de la science par les chercheurs. Le Calcul Haute-Performance (HPC) est la science visant à améliorer les performances des ordinateurs pour répondre aux besoins de la simulation numérique. Son objet d'étude est le supercalculateur : un ensemble de nœuds de calcul interconnectés en réseau, chacun composé de processeurs contribuant en parallèle à la simulation. Au fil des décennies, les superordinateurs ont connu une évolution rapide, avec une augmentation et une diversification des unités de calcul au sein des nœuds. En conséquence, la programmation de codes de simulation numérique portables et performants entre les différentes générations de machines devient un défi.

Pour y répondre, les vendeurs de matériel et les chercheurs conçoivent des modèles de programmation standards tels que « The Message Passing Interface (MPI) » ou « Open Multi-Processing (OpenMP) ». Ces modèles servent de pont entre l'expression d'un calcul scientifique par un code informatique et son exploitation par le matériel sous-jacent. La diversification du materiel pousse les programmeurs à utiliser conjointement (ou « composer ») de multiples modèles de programmation asynchrones afin d'utiliser toutes les unités de calcul du supercalculateur de façon concurrente. La composition de modèles asynchrones présente des difficultés pour le profilage des applications, leur programmation et leur niveau de performances.

Dans cette thèse, nous étudions la composition par tâche des modèles de programmation MPI et OpenMP, et apportons les contributions suivantes en réponse à ces difficultés. Tout d'abord, nous présentons un modèle de performance unifié par tâche pour l'hybridation MPI+OpenMP, en définissant des métriques et en implémentant un profileur pour analyser un programme après son exécution. Ensuite, nous concevons un ordonnanceur unifiant la progression et le recouvrement d'opérations asynchrones et hétérogènes, telles que les transferts de mémoire par le réseau, les tâches de calcul, ou les transferts mémoire/calcul depuis/vers un accélérateur. Enfin, nous évaluons et améliorons l'hybridation par tâche sur des proxy-applications modélisant des simulations scientifiques ; en caractérisant et atténuant des goulots d'étranglement aux performances, tels que l'ordonnancement des communications, la limitation du nombre de tâches en vol, ou la découverte du graphe des dépendances.

Keywords: Calcul Haute Performance, Modèle de Programmation, Tâche, MPI, OpenMP

Abstract

Numerical simulations are a powerful tool for scientific research and the industry, considered the third pillar of science by scientists. High-Performance Computing (HPC) is the science of improving computer performances to meet numerical simulation needs. Testbeds for these studies are supercomputers: a set of interconnected compute nodes, each composed of multi-core processors contributing in parallel to perform a simulation. Over the last decades, supercomputer architectures have been living a fast evolution; we observe an increase and a diversification of processing units per compute node. Hence, programming portable and performing numerical simulations across hardware generations is challenging. To meet this challenge, hardware vendors and programmers from scientific laboratories conceive standard programming models such as « The Message Passing Interface (MPI) » ou « Open Multi-Processing (OpenMP) ». These models serve as bridges between numerical simulation code expression and its exploitation by the underlying hardware. Hardware diversification leads programmers to use multiple programming models jointly (or « compose ») to fully exploit the underlying hardware. Asynchronous programming models composition presents difficulties on profiling, programming and performances.

In this thesis, we study the task-based composition of MPI and OpenMP, providing the following contributions on the mentioned difficulties. First, we present a unified task-based performance modeling of MPI+OpenMP composition, defining performance metrics and implementing a run-time profiler with post-mortem analysis. Then, we conceive a scheduler unifying the progression and the overlap of heterogeneous asynchronous operations, such as network memory transfers, computational tasks, accelerator memory transfers and offloading. Finally, we evaluate and improve the task-based composition on proxy-applications modeling real-world scientific simulations, characterizing and mitigating performance bottlenecks such as communication scheduling, throttling, or dependency graph discovery.

Keywords: High-Performance Computing, Programming Model, Task, MPI, OpenMP

Acknowledgements

J'adresse d'abord mes remerciements aux membres du jury pour l'évaluation de mes trois dernières années de travail. En particulier, merci à Thierry, Patrick et Adrien qui ont dirigé et encadré cette thèse. Vous avez été source de précieux conseils, et êtes restés un appui infaillible au cours de ces trois années. J'ai sincèrement apprécié travailler avec notre petit groupe, et je souhaite à chaque doctorant la chance d'obtenir une telle qualité d'encadrement.

De façon générale, je remercie l'institut du CEA pour l'environnement qui m'a été offert. Je remercie en particulier les membres qui ont participé à ma formation d'ingénieur, puis avec qui j'ai eu le plaisir de travailler. Je pense par exemple à Marc, Julien et Hugo. Merci également à J.B. et Julien pour leur support sur MPC et les repas que nous avons partagé. Finalement, pour leur solidarité et leur (quasi) constante bonne humeur, merci à tous les doctorants et stagiaires de TERATEC avec qui j'ai partagé un bout de parcours. Courage à ceux en rédaction, en préparation de soutenances, et à ceux sur qui ça va bientôt tomber ! Je ne me risquerais pas à tous vous citer, vous vous reconnaîtrez :-)

Je remercie également mes collègues d'Avalon. Malgré la distance et mes rares venus, j'ai toujours été accueilli chaleureusement. Les liens étroits à l'international par le JLESC m'a offert la chance d'échanger avec nombreux de mes collègues chercheurs HPC.

In addition, I would also like to thank the members of the RIKEN Center for Computational Science for the internship I was offered during the summer of 2023. It was an intense month, juggling between experimentations on Fugaku, visits in the many beautiful places in Japan, and ... the writting of this manuscript! It would not have been possible without the very welcoming environment offered by M. Sato, H. Murai and Miwako, whom I thank once again.

Finalement, je remercie mes amis, mes parents, mon frère et ma sœur pour leur soutien indéfectible.

Contents

Part I

Context

Chapter 1

Introduction

Contents

1.1 Motivations

Numerical simulation have become a powerful tool for scientific research and the industry. The use cases are wide: military defense, weather forecasting, chemistry, or vehicle security. In the scientific field, Numerical Simulation is considered the third pillar of science, complementing Theory and Experiment. It is, in particular, necessary for solving non-linear, multi-physics, and large-scale problems [1]. In the industry, it improves the validation of systems; for instance, simulating thousands of car accidents by varying initial conditions helps to characterize material deformation. Their accuracy and speed are sometimes critical: in Japan, severe weather can occur in less than 20 minutes, which motivated the development of simulations to explore weather forecasts on a 100-m grid spacing over 30 minutes to evacuate citizens on time [2]. High Performance Computing (HPC) is the science of improving the *computational performances* to meet the accuracy and speed challenges of numerical simulations.

The high-performance computing object of study is supercomputer: a set of interconnected machines (*compute nodes*) contributing in parallel to progressing a numerical application, such as simulations. The performance of supercomputers are measured by the number of numerical operations they perform per second (FLOPs/s). The TOP 500 list ranks worldwide supercomputers by their computational performances for resolving dense linear systems. It also provides hardware details of each ranked supercomputer. Over the last 20 years, the trend shows the evolution of compute nodes from single-core processors to multi-heterogeneous cores. In 2002, EARTH-SIMULATOR[1] supercomputer was ranked first using 5,120 single-homogeneous core compute nodes for a total of 40 TFlop/s. In 2010, Tianhe-1[2] was the first supercomputer with heterogenous cores (CPUs and GPUs) on the same compute nodes to perform above 1 PFlop/s. As of 2023 writing this thesis, Frontier is now the most powerful supercomputer with compute nodes embedding 64-core processors and 4 GPUs[3] for more than 1 EFlop/s in total.

A numerical application *efficiency* is its ability to fully exploit the underlying supercomputer to progress the computation. It can be quantified by comparing real execution performances over theoretical system performances, or peak performances provided by the TOP 500. The

[1] `https://www.top500.org/system/167148/`
[2] `https://www.top500.org/system/176929/`
[3] `https://www.top500.org/system/180047/`

fast hardware architecture evolution faced over the last 20 years degraded the efficiency of existing simulation code that was originally conceived for single-core architectures: the problem of performance *portability* consists in keeping simulations efficient over past, new and future supercomputers architecture. In order to improve the performance portability of numerical simulation, programmers conceive simulation codes using *programming models*. It consists of an Application Programming Interface (API) providing the code building blocks and an Execution Model (EM) specifying how the executing environment must interpret the API. Therefore, Programming models serve as bridges between a calculation *expression* and its *exploitation* by the underlying architecture.

To ensure coherent bridging, programming models such as the Message Passing Interface (MPI)[4] or Open Multi-Processing (OpenMP)[5] are conceived by a large community of hardware vendors, research laboratories and programmers. Modifications are discussed, voted and eventually adopted in *standard specifications*. Such a collaborative approach to conceiving programming models provides guarantees to numerical simulation programmers on their code performance portability.

The diversity of processing units now imposes on programmers to use multiple programming models jointly, for instance:

- MPI requests to send messages by the interconnection network through the network interface controllers (NICs),
- pthreads or OpenMP to execute code on all available central processing units (CPUs),
- OpenMP for vectorized instructions (AVX, SVE...),
- CUDA/HIP/OpenCL/OpenMP to offload computation to accelerators (GPUs, FPGA),
- MPI-IO/Lustre to store data onto permanent storage (hard drive, magnetic tape).

Following historical hardware evolution, numerical simulation codes had been originally distributed on compute nodes using MPI only; and then parallelized with OpenMP to benefit intra-node parallelism introduced 20 years ago. Usually, computational loops are parallelized using OpenMP (CPUs/GPUs) followed by a local synchronization on the compute-node to wait for computation completion; and only then network communications (NICs) are initiated with MPI, followed again by synchronizations before pursuing computational loops[6]. Such distinct use of the two programming models simulation codes alternatively underload and overload each processing unit with bulk workloads On the other hand, all these processing units are capable of *asynchronous* execution: some iterations from computational loops could execute in parallel with independant network communications (NICs). Using each programming models distinctly, codes are not expressing such asynchrony leading to an inefficient use of the hardware.

The standard community primarily focused on using MPI with OpenMP before the introduction of dependent tasking in 2013. Dependent task-based programming is a promising approach to using multiple asynchronous programming models more tightly, which we refer to as programming model *composition*. It divides applications into tasks (as a sequence of instructions) and precedence constraints providing partial order of execution. Task-based execution models are rather simple: (1) a task can only execute once all its precedence constraints are met, and (2) only one task at a time can run per processing unit. This simplicity allows the representation of other programming models building blocks (MPI requests, CUDA streams) as a set of dependent tasks to be processed seamlessly by a shared execution model: the problem of performance portability is then partly differed to a question of task scheduling.

[4] `https://www.mpi-forum.org/meetings/`
[5] `https://www.openmp.org/about/members/`
[6] `https://asc.llnl.gov/coral-2-benchmarks`

1.2 Problem Statement

Recent advances in the OpenMP dependent task model open new doors to compose programming models. Simulation codes can express a higher potential of asynchrony through smaller workload units (tasks). This approach allows a more performant exploitation of the hardware, staggering tasks on each processing unit over time instead of alternatively overwhelming them with bulk workloads as done historically. Yet, since 2013, few to no applications have migrated towards task-based programming due to three tied difficulties:

- Profiling. Distributed task-based applications execution is asynchronous and out-of-order, making it difficult to understand their behavior. Hybrid profiling and visualization tools (Gantt charts, graphs) are needed [3], but none supports standard MPI+(task-based OpenMP) composition: performance issues can hardly be understood, as reported in [4].
- Programming. When modifying existing program, E. Aubanel [5] mentioned that "*the data flow is not as obvious as the control flow, and the function of the program is the hardest to discover*". In our case, porting an existing program to use dependent tasks requires precise understanding of its data flow, so programmers express how communication and computation tasks are interleaved using tasks and dependencies. In particular, side-effects (or 'interoperability issues') appear when mixing standard programming models implementations in production environments (GCC, LLVM, Open MPI): running MPI communications into OpenMP tasks most likely leads to deadlocks [6]. Hence, the need for programming interfaces to ease programs porting and their maintenance in the future.
- Performances. Task-based application performances are a lot about finding a compromise between the parallelism expression (tasks granularity, dependencies) and induced run-time management costs [7]. They are also about task scheduling: misuse of the memory hierarchy [8, 9] or the network [10] can significantly degrade performances.

1.3 Objectives and Contributions

In this thesis, we investigate and propose solutions to the three difficulties of migrating applications to a task-based programming model composition to improve their performance portability. We use the Open Multi-Processing (OpenMP) tasking model and the Message Passing Interface (MPI), as their collaborative way of working will most likely make them survive in time. Most developments had been made as part of the existing Multi-Processor Computing (MPC) [11] OpenMP tasking runtime. We provide the following contributions to each difficulty:

- On profiling, we propose a unified task-based performance modeling of MPI+OpenMP applications. We define performance metrics implemented as part of a run-time profiler and post-mortem analysis. This toolchain allows precise assessment on the performances of an instance of execution. Ultimately, it leads us to show impacts on the interleaving between the OpenMP task dependency graph discovery and its parallel execution over performances.
- On programming, and to respond the lack of applications using task-based composition of MPI and OpenMP, we ported a benchmark (HPCG) and a proxy-applications (LULESH[7]) to the task-based composition understudy. We provide porting experience feedback and a few extension proposals to OpenMP aiming to ease the porting of existing production simulation codes. We always consider our proposals by reflecting on their impacts on user simulation codes, runtimes and compilers. In particular, we propose the removal of some OpenMP restrictions which currently harden irregular dependencies expression; an

[7] `https://github.com/rpereira-dev/LULESH`

extension for heterogeneous memory address space dependency expression; new mechanisms for tasks suspension/resumption; and an original OpenMP tasks priority management to favor the early-bird posting of tasks performing MPI communications.

- On performances, our developments in MPC have lead to a cutting-edge and open-source OpenMP tasking runtime implementing most recent standard interfaces[8]. Our cooperative task scheduling design shows improved performances on fine-grain tasks offloading to GPU over existing solutions. We also introduce and implement an original tasking extension that drastically reduces fine-grain tasking overheads: task persistence.

A significant contribution also lies in evaluations conducted as part of this thesis: it provides an idea on the level of performance to expect after porting existing simulation code to the task-based composition understudy under current MPI and OpenMP technology readiness.

We organize the manuscript as follows. Part I presents the context of this thesis: Chapter 2 discusses general aspects of numerical simulations (what is / how to program a supercomputer); and Chapter 3 focuses on standard programming models and their hybridization. Part II presents our contributions we organized in three chapters: Chapter 4 presents the unified modeling and profiling tools we built; Chapter 5 presents standard and runtime extensions to improve scheduling capabilities; and Chapter 6 presents the porting of two mainstream HPC benchmarks (HPCCG, LULESH) with standard extensions to ease the porting of future irregular applications, and investigate issues related to the task dependency graph discovery. Finally, Part III concludes and provides research perspectives.

[8] `https://github.com/cea-hpc/mpc/tree/cea/2023/icpp-interop-tasks`

Chapter 2

Numerical Simulations on Supercomputers

Contents

In order to execute on supercomputers, numerical simulations rely on a vast stack built upon years of research and industrial uses. The iceberg Fig. 2.1 is an overview of the current HPC ecosystem, which we propose to divide into four layers. On the lowest layer, the Hardware represents electronic components provided by vendors, historically built through co-design to meet supercomputer application needs. Above, the Operating System corresponds to standard interfaces and the Linux kernel, which equip every supercomputer node to manage the Hardware and provide a primary bridging interface to programmers. Finally, the Software layer, built on top of the operating system, represents programming languages, compilers, libraries, tools, and programming models.

2.1 Supercomputers Architecture

At the lowest layer of the HPC stack, hardware are the electronic components provided by vendors. We describe modern HPC hardware architecture with a zoom on the CEA-HF supercomputer compute nodes, which was recently in 2021-2022 and experimented as part of this thesis.

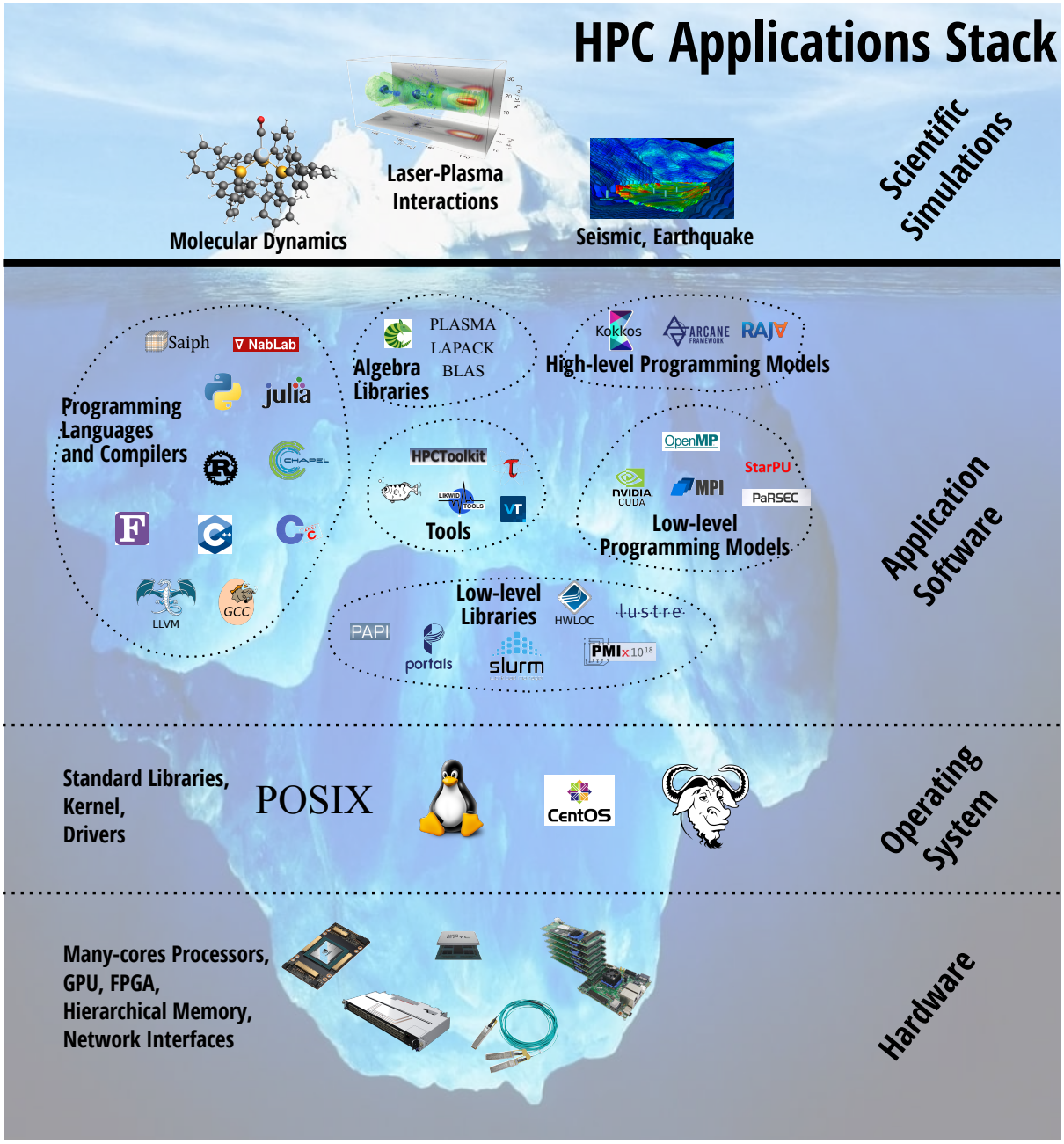


Figure 2.1: Scientific Simulations Application Stack Overview. Top-most "Scientific Simulations" images from [12, 13, 14]

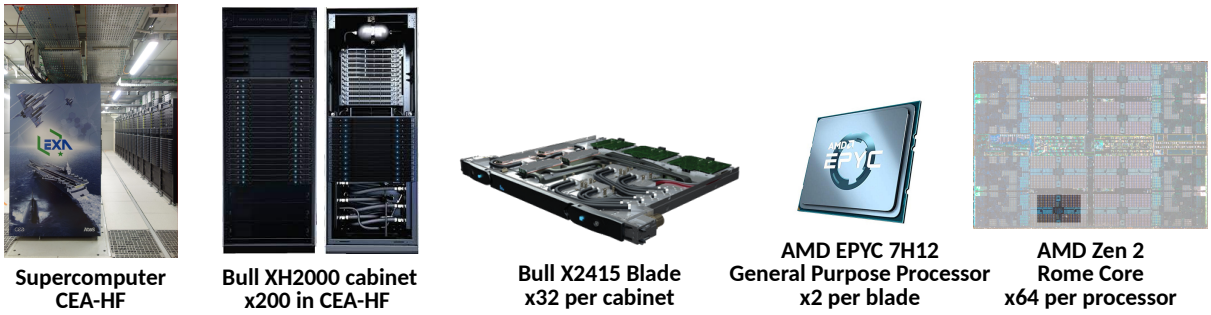


Figure 2.2: CEA-HF Supercomputer Architecture

2.1.1 Introducing the CEA-HF Supercomputer

The CEA-HF supercomputer[1] was installed between 2021-2022 as the successor of TERA-1000 in 2016-2017, ranked 22 on the top500 in June 2023. Fig. 2.2 provides an overview of its architecture[2]. The supercomputer is made of 200 Bull XH2000 42U Cabinet ($1U \simeq 4.5cm$). Each cabinet embeds a power management controller, a hydraulic cooler system; network switches for the interconnection network, and 32 standardized entries for compute blades. 1U compute blade installed in each cabinet is made of 2 AMD EPYC 7H12 General Purpose Processor (GPP) and 1 BullSequana eXascale Interconnect (BXI V2) network interface controller. For comparison, the exaflopian Frontier machine uses the same GPPs on AMD blades accelerated with 4 Instinct 250X GPUs, assembled into Cray cabinet/interconnection.

2.1.2 Zoom on Modern Compute Blades

Supercomputers' compute blades are the building blocks of a supercomputer, made of one or multiple compute nodes interconnected within a network. Each blade may contain multiple processors connected to a shared memory.

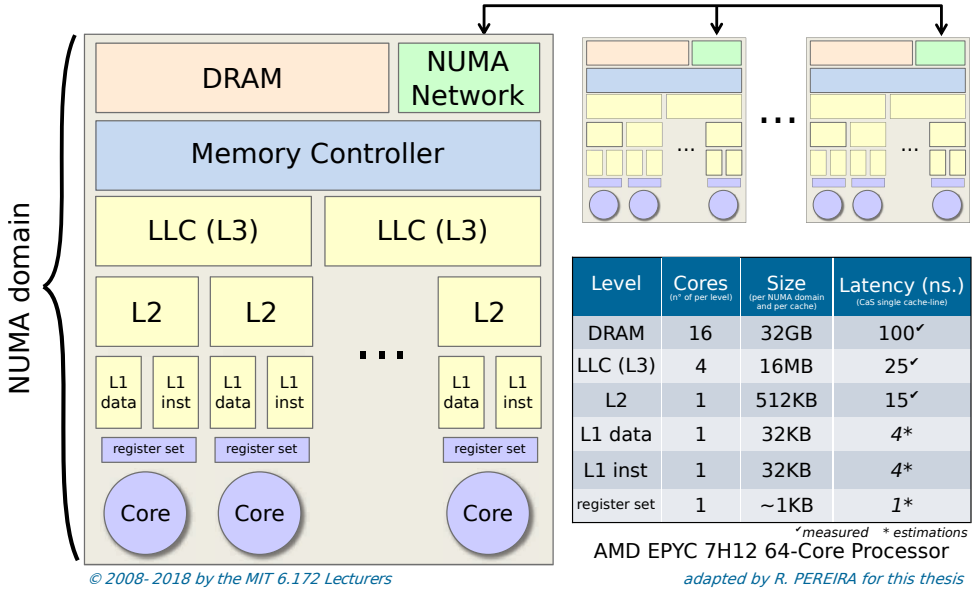


Figure 2.3: General Purpose Processor - Hierarchical Memory

[1] `https://www.top500.org/system/180031/`
[2] adapted from Fig.1.1 of [15] for TERA-1000

2.1.2.1 General Purpose Processor (GPP)

A General Purpose Processor (GPP) is the processing unit equipping every personal computers and supercomputers node. It is designed to perform any numerical operations (additions, multiplications, conditions testing, reading/writing to the memory, sending a signal on a bus...). It is built with multiple cores hierarchically connected to a shared memory.

Hierarchical Memory Fig. 2.3 shows the memory hierarchy of most GPPs (adapted from the lecture [16]). Taking the AMD EPYC 7H12 64-core Processor as an example, each core has its own register set and L1/L2 caches; L3 caches are shared between 4 cores; NUMA domain is made of 16 cores that can access their local DRAM uniformly; and the processor is made of 4 NUMA domains. Each level of the hierarchy does not come with the same size and access latency: the table on the right depicts the core-to-core compare-and-swap latency benchmark[3] on AMD EPYC 7H12 GPP (entire results in annex Fig. 8.1). As mentioned in [17], we observe an "*inverse relationship between the size and access time of computer memories*" that is inherent of the hardware. An interactive history of the memory accesses latency over the past 20 years has been made by C. Scott and is available online[4].

Cache and Prefetching When a core reads from or writes to the DRAM, it first checks if the data is already in a lower cache level. In such case, the processor reads or writes to the cache instead of the DRAM accelerating its memory accesses latency: this is a *cache-hit*. Otherwise, a block of memory from the DRAM is copied *cache lines* (of fixed size, typically 32, 64, or 128 bytes[5]): this is a *cache-miss*. Cache *prefetching* [17] consists in bringing memory from the DRAM to caches before it is required by anticipating memory access pattern. Such techniques reduce memory accesses latency by overlapping independent instructions with DRAM fetching into caches. Prefetching can be implemented in the hardware and by software (done mainly by compilers currently). A prefetching decision may be *accurate* or *inaccurate* depending if the prefetched memory will actually be accessed by cores or not. Several research works have been conducted on reducing inaccurate prefetching predictions, as they may pollute caches erasing useful cache lines previously fetched, but there is none universal to any architecture and applications [18].

Instructions Pipelining As soon as a GPP is booted, its cores start processing instructions until they are physically shut down. An *instruction* is an opcode and a set of operands. Opcode and operands are integers coding an operation to be performed by the architecture: the opcode defines which operation, and operands are its parameters. The classic RISC instruction processing is a five-stage: `fetch` > `decode` > `execute` > `memory access` > `write-back`[6]. Pipelining performances are measured as the number of Instructions performed Per Cycle (IPC): in an ideal single RISC pipeline, the IPC is 1 (with one `write-back` stage performed per cycle). In practice, the IPC of a RISC pipeline is lower than 1 due to *hazard*: situations where it is necessary to prevent the processing of an instruction stage to ensure correctness of the operation (Chapter 2 of [19]). For instance, a data missing into cache memory may lead to an hazard, and sometimes a *stall* blocking entirely the core pipeline until the data arrives.

Cores Modes In addition, each core of a GPP can run multiple *modes*. For instance, the x86 architecture provides 4 modes (0 is *kernel*, 3 is *user*). In the kernel mode, cores can perform any instructions implemented by the architecture: it may read/write every memory byte or even shut

[3] `https://github.com/nviennot/core-to-core-latency`
[4] `https://colin-scott.github.io/personal_website/research/interactive_latency.html`
[5] `http://www.nic.uoregon.edu/~khuck/ts/acumem-report/manual_html/ch03s02.html`
[6] `https://en.wikipedia.org/wiki/Classic_RISC_pipeline`

down hardware components. In the user mode, cores can only perform a subset of instructions with limitations on their operands, as restricted operations could lead to irreversible damage to the machine.

2.1.2.2 Network Interface Controller (NIC)

A Network Interface Controller (NIC) is a hardware component with its own memory and processing units, typically connected to the GPP memory through a PCI bus. A compute blade can embed multiple NIC per GPP, or multiple GPP per NIC: in any case, there is to interconnect the compute blade of the supercomputer. This interconnection network allows any compute nodes to communicate memory to one another, and the NIC manages transfer protocols. In particular, NICs are implementing Remote Direct Memory Accesses (RDMA) protocols [20], allowing concurrent accesses to the memory with GPP cores. RDMA enables zero-copy (to/from GPP/NIC) communications from pinned memory regions, reducing transfer latency. In addition, NICs now also embed processing units, partly removing the need for GPP clock cycle to progress communications.

2.1.2.3 Graphic Processing Unit (GPU)

A Graphics Processing Unit (GPU) is a specialized processing unit used in HPC for its energy efficiency on dense linear algebra. On the Green500 ranking, every supercomputer on the top 10 rankings uses GPUs, and performs with about 40 to 60 Gflops/watts. On the other hand, the Chinese supercomputer Tianhe-2A ranked 10 on the Top500 is performing at 3 GFlops/Watt with a pure-GPP architecture, which is one order of magnitude less energy-efficient than GPU machines. This energy efficiency made it famous in the HPC industry and ultimately led to the Frontier exaflopian machine under 20MW consumption: a power restriction set by the US Department of Energy (DoE) in 2009 when power projection using existing technology where estimating 300MW to reach exaflopian performances then [21].

2.1.2.4 Summary

In order to reach high performances on supercomputers, the diversity and complexity of hardware must be taken into account. Our contributions in Chapters 4,5,6 primarily focus on GPP-only compute nodes interconnected in network from their NICs, such as the CEA-HF supercomputer. Results in Section 6.2 illustrate how an improved use of the memory hierarchy and caches can reduce pipeline stalls and double computational performances. While it is not presented in this document, preliminary results on accelerating the CEA-HF architecture with GPUs had been published in [22].

2.2 Executing and Programming on a Supercomputer

Climbing up the scientific simulation iceberg Fig. 2.1, the Operating System (OS) is the lowest software layer of a computer. It is responsible of managing presented hardware by providing simpler and portable interfaces for upper-layer application software such as OpenMP. The Portable Operating System Interface (POSIX) [23] specifies standard interfaces for operating systems.

2.2.1 The Portable Operating System Interface (POSIX)

The Open Group is a worldwide consortium of 892 members at the time of writing this thesis[7] including banks, IT companies (Fujitsu, Intel, Huawei...) or even governmental institutions

[7] `https://opengroup.org/aboutus`https://opengroup.org/aboutus

(Canadian Department of Defense, research institutes...)[8]. Its mission is to provide open and vendor-neutral technology standards and certifications, and in particular, it provides the POSIX specifications defining operating system concepts used in every implementation to some extent (MS-Windows, GNU/Linux, Mac OS, NetBSD, ...). For instance, the 2017 specifications provide the following definitions referenced in this thesis:

Program "*A prepared sequence of instructions to the system to accomplish a defined task*"

Thread "*A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, errno value, floating point environment, thread-specific key/value bindings, and the required system resources to support a flow of control. [...]*"

Live Process "*An address space with one or more threads executing within that address space, and the required system resources of those threads.*"

Address Space "*The memory locations that can be referenced by a process or the threads of a process.*"

The GNU/Linux operating system mostly complies with the POSIX standard, and above definitions, and is equipping every compute node of every supercomputer ranked on the Top 500 in the November 2022 list[9]. Therefore we introduce its Linux *kernel* briefly.

2.2.2 Operating System Kernels: a Fast Zoom on Linux

When booting a computer, the operating system *kernel* is one of the first programs executed. It remains in a protected memory region until the computer shutdowns and is a critical piece of software in terms of security and performance. Security-wide, cores process the kernel instructions in the kernel mode, giving it entire access to the system resources. In addition, the kernel is responsible for impactful mechanisms such as *interrupts*, *multi-process scheduling*, or *memory management* that can have a significant impact on performances. As the *Linux* kernel [24, 25] runs on every supercomputer ranked on the Top 500, we briefly present a few of its components that we reference later in this thesis.

2.2.2.1 Interrupts

Interrupts are a means of communication between hardware components and the kernel. This mechanism was originally designed to ensure responsiveness between hardware interactions (pressing a key) and the operating system (writing the letter on the terminal). Each interrupt has an identifier that maps to an *handler* through an Interrupt Descriptor Table (IDT). When an interrupt occurs, the CPU suspends its execution flow and switches to its associated handler program. In the Linux Kernel Development guide [24], interrupts are presented as an alternative with less impact on performance than periodical execution flow suspension and polling, as an interrupt suspension is attached to an actual hardware event.

However, in the context of data centers, G. Regnier et al. [26] evaluated the operating system footprint (system call, interrupts, memory copies) to represent 50% of TPC/IP communications processing. This leads the HPC community to heavily optimize this part of the software stack, particularly with the implementation of Direct Memory Accesses (DMA) to reduce kernel involvement in communications. Using the Linux DMA driver API[10], interrupts and memory copy costs can be removed, as hardware components (NICs, GPUs...) can directly write into

[8] `https://reports.opengroup.org/all.shtml`
[9] `https://www.top500.org/`
[10] `https://docs.kernel.org/driver-api/dmaengine/client.html`

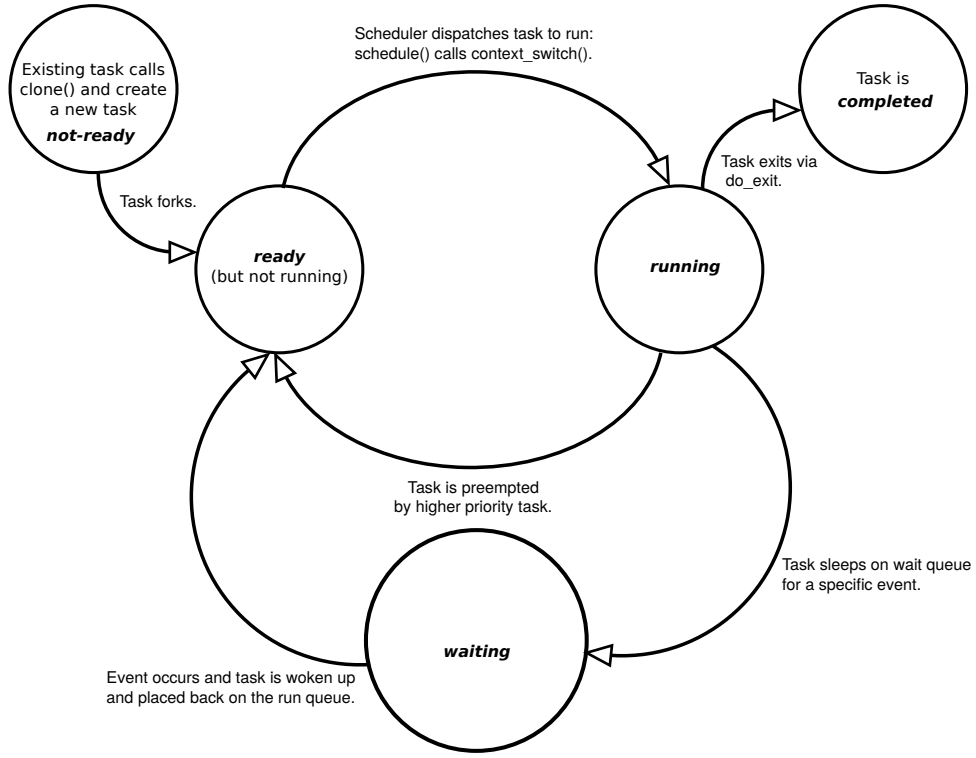


Figure 2.4: Linux `task_struct` states adapted from [24]

the user-space process memory. In parallel and asynchronously of a DMA, cores eventually poll the memory and may notice the completion of a request asynchronously without interrupting execution. Note that DMA sometimes requires memory to be pinned (i.e., virtual memory mapping to physical memory is constant); but not always: for instance, Bull BXI NICs can translate virtual to physical addresses [27].

2.2.2.2 System Calls

System calls are interfaces provided by the operating system, so user-space programs can interact with the kernel. As user-space programs cannot execute kernel code directly, Linux implements system calls as an interruption with parameters passed using registers (instruction `int x80`, and registers `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp` on x86). The kernel implements a handler that fall-backs to the associated kernel-space implementation of the system call.

Note that the cost of a system call is significantly higher than a traditional function call. In [28], authors conducted several experiments on Intel Nehalem (Core i7) to quantify costs on (1) switching processor mode ($\sim$ 150 cycles), (2) memory pollution (caches) and (3) impacts on IPC. Their results show that a system call is at least an order of magnitude higher than a conventional x86 user-space function call.

2.2.2.3 Thread/Process Management and Scheduling

Linux provides interfaces for managing POSIX threads and processes. Internally, the kernel represents both threads and processes using the `struct task_struct` data structure sizing about 1.7KB on a 32-bit machine[11]. Fig. 2.4 depicts the five states of a Linux `task_struct`: it can whether be *not-ready*, *ready*, *running*, *waiting* or *completed*. Each instance of a `struct task_struct` is forked through the `clone` system call and may share attributes (e.g. virtual

[11] `https://github.com/torvalds/linux/blob/a901a3568fd26ca9c4a82d8bc5ed5b3ed844d451/include/linux/sched.h#L738-L1549`

memory mapping, file descriptors...). They embed multiple execution contexts (*i.e.* a stack and registers copy): a kernel-mode execution contexts for executing kernel code and interruptions; and a user-space execution context for executing user code. In this thesis, *Kernel-Level Threads (KLT)* refers to Linux `task_struct` and *User-Level Threads* refers to user-level execution contexts.

Linux `task_struct` instances can be created by threading libraries implementations such as the GNU POSIX threads (pthreads)[12]; a POSIX thread is created as such:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Scheduling-wise, since 2007, Linux schedules its `task_struct` on CPUs using the Completely Fair Scheduling (CFS) scheduler implementation. In shorts:

- Tasks are sorted by their *virtual run-time*; a metric that is built upon their run-time on physical cores (time spent in the *running* state).
- When returning from the kernel code to user-space (after the completion of a system call or an interrupt handler), the kernel picks the *ready* task with the least virtual run-time executes it.

In addition, the Linux kernel is configured on boot with a time period at which a *timer* interrupt is raised, *preempting* the execution of the current task. The timer interrupt handler is defined in the `kernel/time/tick-common.c` file[13]. It updates various timers, and in particular: the *current task virtual runtime*. Hence, before returning, and just like any other interrupt handler, the kernel may decide to switch tasks.

2.2.2.4 Summary

The GNU/Linux operating system is mostly compliant with POSIX, providing portable interfaces on every supercomputer listed on the top500. As the lowest software layer of the HPC application stack, the OS manages the hardware, making it a critical component for performance. The operating system overheads on interruptions, system calls, and periodic preemption guided our research towards user-level thread scheduling: threads with no existence for the Linux kernel, running in the user-mode of CPUs. Section 5.1 presents OpenMP tasking extensions to enhance its scheduling flexibility with user-level threads.

In the next sections, we climb-up the HPC application stack towards the "Application Software" layer. We first provide a brief history on HPC programming languages and presents existing technologies.

2.2.3 Programming Languages and Compilers

Programming is the action of creating a program, that is, writing down a sequence of instructions to perform a specific task. Early in the 1950s, programming consisted in writing a sequence of CPU instructions opcode and operands. This is depicted on Listing 2.1 using x86 GNU Assembly (that could be directly converted to opcode/operands machine code), whose task is to compute 4×5 and output the result. Such a programming approach presented major drawbacks: programs are difficult to conceive, maintain and understand, and they needed to be more portable across processor architectures. J. Backus [29] mentions that back then, low-level programming was not economically sustainable as programmer costs were "*at least as great as the cost of the computer itself [...] [and] as computers got cheaper, this situation would get worse*".

This lead to the development of the FORTRAN programming *language* and *compiler* by J. Backus team at IBM in 1954. The programmer write a program in a language (FORTRAN) agnostic of the processor architecture, and the compiler is responsible of converting it to an

[12] `https://www.gnu.org/software/libc/manual/html_node/POSIX-Threads.html`

[13] `https://github.com/torvalds/linux/blob/a901a3568fd26ca9c4a82d8bc5ed5b3ed844d451/kernel/time/tick-common.c#L107-L147`

```
section .data
num1 dd 4
num2 dd 5
format db "%d", 0
result dd 0

section .text
global _start

_start:
    mov eax, [num1] ; move '4' from the data segment to the 'eax' register
    mov ebx, [num2] ; move '5' from the data segment to the 'ebx' register
    imul ebx        ; compute '4 x 5 = 20' to the 'eax' register
    mov [res], eax ; move '20' from the 'eax' register to the data segment

    ; call printf to print the result
    push dword [res]
    push dword format
    call printf
    add esp, 8      ; remove arguments from the stack

    ; exit the program
    mov eax, 1      ; 'sys_exit' system call
    xor ebx, ebx    ; set the system call argument to 0
    int 0x80        ; raise the system call
```

Listing 2.1: X86 Assembly program

architecture-specific low-level program (assembly, machine-code) so the hardware can execute it. Listing 2.2 illustrates a FORTRAN program that could be compiled to the Assembly code Listing 2.1. Abstracting programs from architecture through programming languages and compilers tackle machine-code programming drawbacks. It has led, since then, to significant development of programming languages and compilers.

```
program multiplication
implicit none
integer :: num1, num2, res

! Multiply 4 x 5
num1 = 4
num2 = 5
res = num1 * num2

! Print the reuslt
write(*,*) res

end program multiplication
```

Listing 2.2: Fortran program

```
int main(void) {
    int num1 = 4;
    int num2 = 5;
    int res = num1 * num2;
    printf("%d", res);
    return 0;
}
```

Listing 2.3: C program

```
num1 = 4;
num2 = 5;
res = num1 * num2;
print("%d", res);
```

Listing 2.4: Python program

2.2.3.1 A Brief History on Programming Languages and Compilers in Scientific Simulations

FORTRAN (1954), C [30] (1970), C++ [31] (1985), Python [32] (1991) are the top 4 programming languages used by HPC programmers. The Gordon Bell Prize [14] has been awarded each year since

[14] https://awards.acm.org/bell

1987 by a committee from the Association for Computing Machinery (ACM). It awards HPC applications (scientific simulation, engineering, or large-scale data analytics) showing notable performances and portability. Most Gordon Bell papers do not provide source code, and for some of them, too few implementation details to conclude on the programming languages used. Still, we observe the following trends. Between 1987 and 2000, we observe the domination of Fortran in the Gordon Bell Prize awards. After 2000, simulation programming languages start transitioning from Fortran to C/C++, with both languages being about equally represented. On the recent winners (2015 to 2022), simulations are whether using pure C/C++ [33, 34], or a mix of C/C++/Python [35, 36] using C/C++ as low-level programming languages for computations kernel, and Python as a higher-level programming language to orchestrate computational kernels.

C/C++ The C programming language was initially conceived at the Bell Laboratories by D. Ritchie, who presents the language development history in [37]. The language was motivated to create the Unix operating system commands and kernel. It showed to be portable across hardware, which led to the development of compiler support on most existing architectures at the same time, and its use was extended from operating system to application development. Listing 2.3 illustrates our minimal code example using the C programming language. The C++ programming language was initially developed at Bells Laboratories by B. Stroustrup to create a successor to C providing additional facilities[15]. Since then, the language has shown stable evolution and support by compilers. B. Stroustrup is no longer part of the language committee but is still influential. He recently (2021) expressed in an interview[16] that in the future, C++ could evolve toward better integration of heterogeneous processing units programming (CPUs, FPGA, GPU), in particular using hardware CPU/GPU uniform address spaces to relieve the burden of data movement from the programmer, as the hardware would embed implicit support.

The Gnu Compiler Collection (GCC [38]) and LLVM's Clang [39] currently are the two dominating C and C++ compilers. GCC first release was in 1987, and LLVM in 2007, whose motivations were primarily to provide more flexibility (with Just-In-Time compilation, or cross-file optimization, for instance) and faster compilation than the production compiler of that time. Both GCC and Clang are Free and Open-Source Software (FOSS) supporting most of the latest C/C++ language features. They are respectively distributed under the GNU GPL license and the Apache License 2.0. The Apache license allows a close-source and commercial redistribution of LLVM. When selling a machine, vendors would typically sell a compiler (ICC for Intel, AOCC for AMD) that is a patched clone of Clang, finely tuning the compiler for the targeted hardware.

Python As opposed to Fortran/C/C++ implementations, the Python programming language main implementation (CPython[17]) relies on an *interpreter*. It means that on each execution, a Python program is converted (or interpreted) to machine-code while this conversion is only done once for all at compilation-time with Fortran/C/C++ programs. Moreover, the Python language does not support concurrent multi-threading due to shared responsibility between standard definition and runtime implementation[18]. Hence, at first sight, Python sounds like a poor candidate for HPC programming. However, the language is excessively simple (as shown on Listing 2.4) and modular: it can be extended with high-performant C/C++ libraries doing the critical scientific simulation (parallel) computation to be called by the Python interpreter through (sequential) interfaces. According to the TIOBE index[19], the language's popularity kept increasing over the last 20 years, making it the most popular language in 2022.

[15] `https://www.stroustrup.com/1st.html`

[16] `https://www.youtube.com/watch?v=Bycec3UQx0c`

[17] `https://github.com/python/cpython`

[18] In order to enable concurrent multi-threading, the Python Enhancement 703 (PEP703) proposes the removal of the global lock from the CPython runtime implementation, that restricts the interpretation of Python code to a single thread at a time. The Python community is (in 2023) still actively discussing.

[19] `https://www.tiobe.com/tiobe-index/`

2.2.4 HPC-oriented Programming Languages

Many more programming languages have been created specifically for the development of scientific simulations but are, in practice, less represented than Fortran/C/C++/Python.

High Performance Fortran (HPF) The High-Performance Fortran (HPF) [40] was proposed in 1992 as an extension to Fortran for distributed machine programming, but failed to become popular mainly due to immature technology and a lack of efficient implementations [41]. Nevertheless, HPF failure led to important research on *Partitioned Global Address Space* (PGAS) programming, with for instance: X11, Chapel, Fortress, SHMEM, Dash, XCalableMP (XMP). PGASs provide a global memory address space logically partitioned between processing units. Even though the memory is physically distributed, it can be accessed by the program as if it was shared. The execution parallelization and data transfers are managed implicitly from the partitioning.

Julia is a recent programming language released in 2012. V. Churavy et al. [42] describe the language as a bridging alternative between two communities: the scientific simulation programmers using low-level programming languages (C/C++/Fortran) and the data analysts using high-level programming languages (Python/Matlab). Julia uses the LLVM toolchain, and its implementation enables online interpretation and compilation to machine code, but also just-in-time compilation. Its syntax is close to scripting languages like Lua or Python, but as a compiled language, its performances can approach C/C++. As a recent programming language, its ecosystem is still immature and lacks stability. Only time will tell if its popularity in the HPC field will tackle these drawbacks.

Domain Specific Programming Languages (DSL) are, as their name suggests, programming languages dedicated to a certain *domain* of problems. For instance, NabLab [43] and Devito [44] target simulation over grids/meshes using finite-differential equation solvers, and PPML [45] target hybrid particle-mesh simulations. As opposed to general-purpose programming languages (GPL) (C, C++, Python, ...), domain-specific programming languages can restrict the language. It provides a few specific interfaces tied to the domain so that simulation programmers can focus on the science and not on any architectural aspects. DSL programming is often close to mathematical notations familiar to scientific code programmers, as shown on Listing 2.5 which is a NabLab program.

```
1 ComputeLjr: ∀ c ∈ cells(), ∀ n ∈ nodesOfCell(c), res{c,n} = 4 * 5
```

Listing 2.5: NabLab program line of code, computing 4 x 5 on every mesh node

2.2.5 Programming Models in HPC

Programming Models (PMs) are programming language extensions defining an Application Programming Interface (API) and an Execution Model (EM). APIs provide the building blocks for calculation expressiveness to programmers, while the EM specifies how the executing environment must interpret the API. PMs adds an additional layer on top of programming languages, and serve as bridges between a calculation expression and its execution by the underlying hardware in a portable manner over hardware architectures. In the HPC application stack iceberg Fig. 2.1, a programming model is *low-level* if it is self-sufficient (*i.e.* not dependent on any others PMs), and *high-level* otherwise (*i.e.* built upon low-level PMs).

Low-level programming models include for instance OpenMP [46], OpenCL [47], MPI [48], CUDA[20] and HIP [21]. OpenMP and OpenCL primarily focuses on portability at the compute-

[20] `https://docs.nvidia.com/cuda/cuda-runtime-api/`
[21] `https://github.com/ROCm-Developer-Tools/HIP`

node level; that is, ensuring programs compatibility and performances for any hardware vendor, from a single CPU to a multi-GPP / multi-GPU compute node. CUDA is maintained by the Nvidia vendor to program their GPU hardware. HIP is maintained by the AMD vendor with a CUDA compatibility, so it can run on both Nvidia and AMD vendors hardware. MPI provides standardization for communications between compute nodes through the interconnexion network of supercomputers; it is the most widely adopted PM for distributed parallelism in HPC applications.

To improve the portability of codes and performances, higher-level programming models such as Raja [49], Kokkos [50] or Arcane [51] were developed. They provide higher abstracted interfaces which fallbacks to different lower-level programming models such as OpenMP, OpenCL, CUDA, HIP or MPI.

2.3 Task-based Programming Models

Many programming models falls under the category of *Task-based* Programming Models (TPMs) [46, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62]. We introduce a common set of definitions for TPMs, and then we retrieve and extend the taxonomy on task-based programming model API from P. Thoman et al. [63].

2.3.1 A Definition for Task-based Programming Models

Transition Systems were introduced by R. M. Keller [64] (1975) for modeling asynchronous parallel computation. He defines transition system as a pair $(S, \rightarrow)$ where S is a set of state and $\rightarrow$ a binary relation on S, which can be represented as a directed graph with the nodes S and the edges $(s, s') \in S^2$ if and only if $s \rightarrow s'$. Task-based Programming Models (TPM) are parallel and concurrent programming models which mostly comes with a single *task* transition system. Tasks are attached a program and precedence constraints, such as the completion of other tasks. In its simplest form, tasks can be represented as a three-state transition system: *not-ready*, *ready* and *completed* as shown on Fig. 2.5. It transitions from *not-ready* to *ready* state once all its precedence constraints are fulfilled and from *ready* to *completed* state after the sequential execution of its program instructions.

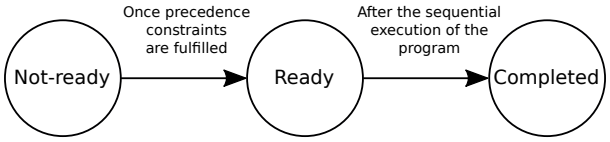


Figure 2.5: A Task Transition System

Task-based programming models usually represent relationship between tasks using graphs, where nodes represents tasks and edges relation between one another. In particular, the Task Control Flow Graph (TCFG) and the Task Dependency Graph (TDG) are commonly used.

Task Control Flow Graph (TCFG) During its program execution, a task T_1 (the parent) may instantiate a new task T_2 (the child). The parent/child binary relation between tasks can be represented as a Direct Acyclic Graph (DAG): this is the *Task Control Flow Graph*. Fig. 2.6 illustrates a TCFG using OpenMP; for instance, (C) and (D) tasks are children of the task (A), as they are created as part of its instruction flow.

Task Dependency Graph (TDG) If the completion of a task T_1 is a precedence constraint for the task T_2, then T_1 is a *predecessor* of T_2, and T_2 is a *successor* of T_1. The predecessor/successor

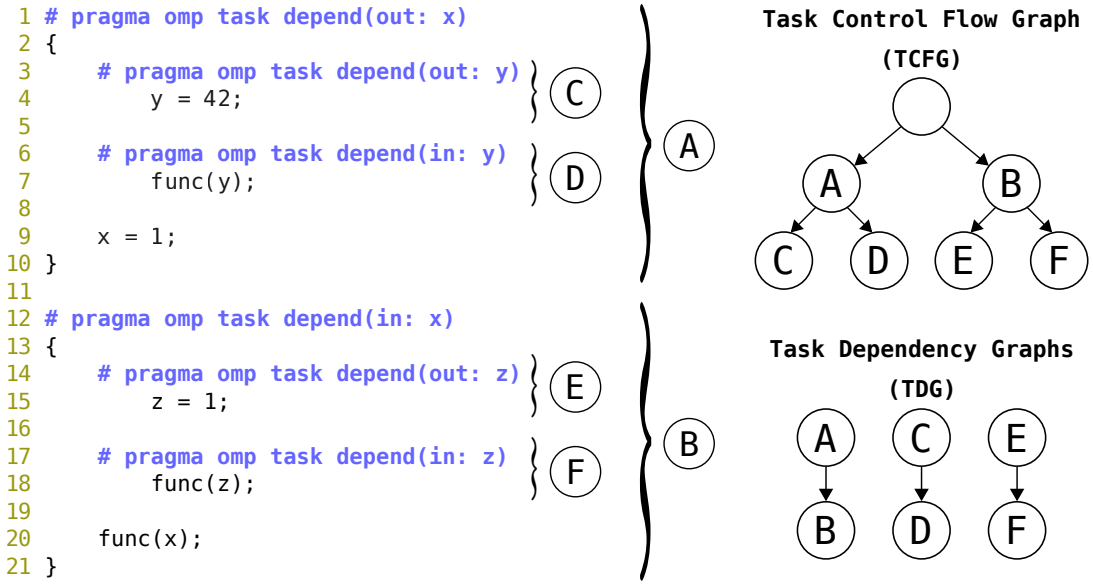


Figure 2.6: Task Control Flow and Dependency Graphs

binary relation between tasks can also be represented as a direct acyclic graph: this is the *Task Dependency Graph*.

2.3.2 A Taxonomy of Task-based Programming Models

Many task-based programming models have been developed. The wide range of their features and applicability motivated P. Thoman et al. to propose a taxonomy of TPM APIs [63]. However, it has a few limitations which lead us to propose a slightly extended version.

2.3.2.1 P. Thoman et al. Taxonomy

Fig. 2.7 depicts P. Thoman et al. taxonomy from their original publication [63]. They considered an important set of TPMs and made a classification out of 13 multi-value parameters. The C++ Standard Library (STL)[22] is considered as a TPM through the `std::async` interface. The Threading Building Blocks (TBB) [52] is a parallel library provided by Intel. OpenMP [46] is a 25 year old general parallel programming language extension for Fortran/C/C++; originally targeting shared-memory architectures and providing a TPM since 2008. ParSEC [53] and StarPU [54] are two libraries providing standard C interfaces for task-based programming. Legion [55] or HPX [56] are TPM executing in a Global Address Space (GAS). Cilk Plus [57], Chapel [58], or X10 [59] are programming languages with their own compilers.

However, the original taxonomy has a few issues:

- The "Graph Structure" parameter specifies "*The type of task graph dependency structure supported by the given API*" which could be a tree structure, an acyclic graph (dag), or an arbitrary graph. This parameter is probably not relevant, as a graph is simply the representation of a binary relation between tasks. For instance, a CFG is always a DAG, and TDGs are mostly DAGs as well even though D. Alvarez et al. [65] recently suggested cyclic TDGs in OmpSs.

- The AllScale project seems abandonned since its European Project ended in September 2018[23] and there is no clear documentation on its API accessible online anymore.

[22] `https://en.cppreference.com/w/cpp/thread/async`
[23] `https://cordis.europa.eu/project/id/671603`

gas	global address space			dag	direct acyclic graph			e	explicit					
smem	shared memory							i	implicit					
msg	message													

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Result Handling	Task Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	dag	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	tree	×	i	✓	i	×	i	i	8	
HPX	gas	i	e	dag	✓	e	✓	i/e	×	i/e	e	6	
Legion	gas	i	e	tree	✓	e	×	i	×	i/e	e	4	
PaRSEC	msg	e	e	dag	×	e	✓	i	✓	i/e	i	4	
OpenMP	smem	×	i	dag	×	i	✓	e	×	i	i/e	9	Extension
Charm++	gas	i	e	dag	✓	i/e	×	i	✓	i/e	e	6	
OmpSs	smem	×	i	dag	×	i	×	i	✓	i	i/e	5	
AllScale	gas	i	i	dag	✓	i/e	×	i	✓	i	i/e	3	
StarPU	msg	e	e	dag	✓	i	×	i	×	i/e	e	5	
Cilk Plus	smem	×	×	tree	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	dag	✓	i	×	i	×	i/e	e	5	
X10	gas	i	i	dag	✓	i	×	i	✓	i/e	e	5	

Figure 2.7: Task-Based Programming Models API Taxonomy from [63]

- StarPU define itself as a "*task programming library*" and not a programming language extension[24] (On "Implementation Type" parameter value).
- OpenMP does provide a *distinct* interface to execute a task on a specific hardware but was still classified as "implicit heterogeinity".
- OpenMP has both implicit (`task` construct) and explicit (`device` clause) work-mapping.
- OpenMP can support "Distributed Memory" to a certain extent; its lack of support is more of related to implementations than its API issue. For instance, A. Patel and J. Doerfert [66] proposed computation offloading from a source node to a target remote (distributed) nodes, without modifying OpenMP API.
- OpenMP does not prohibit implicit "Worker Management" to implementers; for instance, the LLVM implementation does have hidden helper threads [67], that are workers capable of executing specific tasks even outside a `parallel` region.

2.3.2.2 Extending the Taxonomy

We fixed the P. Thoman et al. original taxonomy and extended it with NESL [68], Athapascan-1 [61], XKaapi [60], CPP-Taskflow [62] and MPI. NESL is a programming language designed for parallel and vector supercomputers. Athapascan-1 is a C++ language extension and library for task-based programming with data-flow synchronizations. XKaapi is a C library for fine-grain data-flow synchronizations. It also comes with a C++ library interface and support for the Athapascan-1 C++ extensions. CPP-Taskflow is a C++ TPM library that gained important

[24] `https://starpu.gitlabpages.inria.fr/`

gas global address space
smem shared memory
msg message

e explicit
i implicit

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeinity	Dependency Support	Partitionning	Result Handling	Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	×	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	×	×	i	✓	i	×	i	i	8	
HPX	gas	i	e	✓	✓	e	✓	i/e	×	i/e	e	6	
Legion	gas	i	e	✓	✓	e	×	i	×	i/e	e	4	
PaRSEC	msg	e	e	✓	×	e	✓	i	✓	i/e	i	4	
StarPU	msg	e	e	✓	✓	i	×	i	×	i/e	e	5	
XKaapi	smem	×	e	✓	✓	i	×	i	✓	i	i	7	
CPP Taskflow	smem	×	e	✓	×	i	×	i	×	i	e	8	
MPI	msg	e	e	×	×	e	✓	e	×	e	e	9	
OpenMP	smem	×	e	✓	×	i	✓	i/e	✓	i/e	i/e	9	Extension
Charm++	gas	i	i	×	✓	i/e	×	i	×	i/e	e	6	
OmpSs	smem	×	i	✓	×	i	×	i	✓	i	i/e	5	
Athapascan/Kaapi	smem	i	×	✓	✓	i	×	i	✓	i/e	i	4	
Cilk Plus	smem	×	×	×	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	✓	✓	i	×	i	×	i/e	e	5	
X10	gas	i	i	×	✓	i	×	i	✓	i/e	e	5	
NESL	gas	i	×	×	✓	e	×	i	×	i	i	4	

Figure 2.8: Task-Based Programming Models API Taxonomy adapted from [63]

traction[25]. We also removed the AllScale entry due to it lack of documentation; and we replaced the "Graph Structure" with a "Dependency Support" parameter. It characterizes whether the API support fine-grain synchronization between tasks using dependencies or not (*i.e.* whether the API support the expression of task dependency graphs).

The complexity of the taxonomy shows the diversity of existing TPM and the difficulty of sorting them out. They all come with different features, often as a trade of between performances and automation for programmers

2.4 Conclusion

In this chapter, we briefly presented every layer that constitute the HPC applications stack. We introduced the diversity and levels of parallelism provided by hardware vendors building supercomputers. We presented the programming and execution of HPC applications over supercomputers: the operating system, programming languages, and programming models. They all provide a piece of the answer to the code and performance portability across hardware architectures. In particular, the task-based programming model is gaining traction, one argument being their composability with one another, which enables full exploitation of the underlying hardware. An important set of task-based programming models had been developed in the past, as shown by the discussed taxonomy. In the midst of these numerous existing programming models, *standards* such as the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) try to bring order.

[25]Over 8k stars on the public repository: `https://github.com/taskflow/taskflow`

In this thesis, we focus on the C/C++ programming language extended with MPI and OpenMP and compiled with GCC/Clang on a GNU/Linux operating system. This programming environment is reasonably common in the HPC community. We study the benefits of mixing the two standard to conceive programs taking advantages of both the shared and distributed memory parallelism. The next chapter presents furthermore each programming model individually and their mixed-used using tasks.

Chapter 3

Hybridizing Standard Programming Models

Contents

In the midst of the numerous existing programming models, *standard* programming models such as the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) try to bring order. The *standard* designation means they are designed jointly by hardware vendors and research laboratory. Such a collaborative approach aims to ensure code portability over past, current and future architectures, often through co-design between hardware and software. It also aims to maintain a certain level of performances, referred in the literature as the problem of *performances portability* [69]. Over the last 20 years, programmers had been hybridizing (or "mixing" [70]) the two standards as they both provide complementary benefits to entirely exploit supercomputers. We present each programming models and hybridization attempts proposed in the literature.

3.1 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) is an asynchronous, concurrent and parallel programming model. It provides interfaces for process communication over a supercomputer interconnection network. In [46], L. Dagum mentions that "*Message passing is the native model for these architectures, and developers can only build higher-level models on top of it.*".

MPI Processes, Program, Communications Within the MPI standard, an *MPI process* can be a POSIX process or thread depending on the implementation executing a (POSIX) program. An *MPI program* is a set of *MPI processes* executing the same POSIX program, that may communicate to one another through the MPI API such as *requests*.

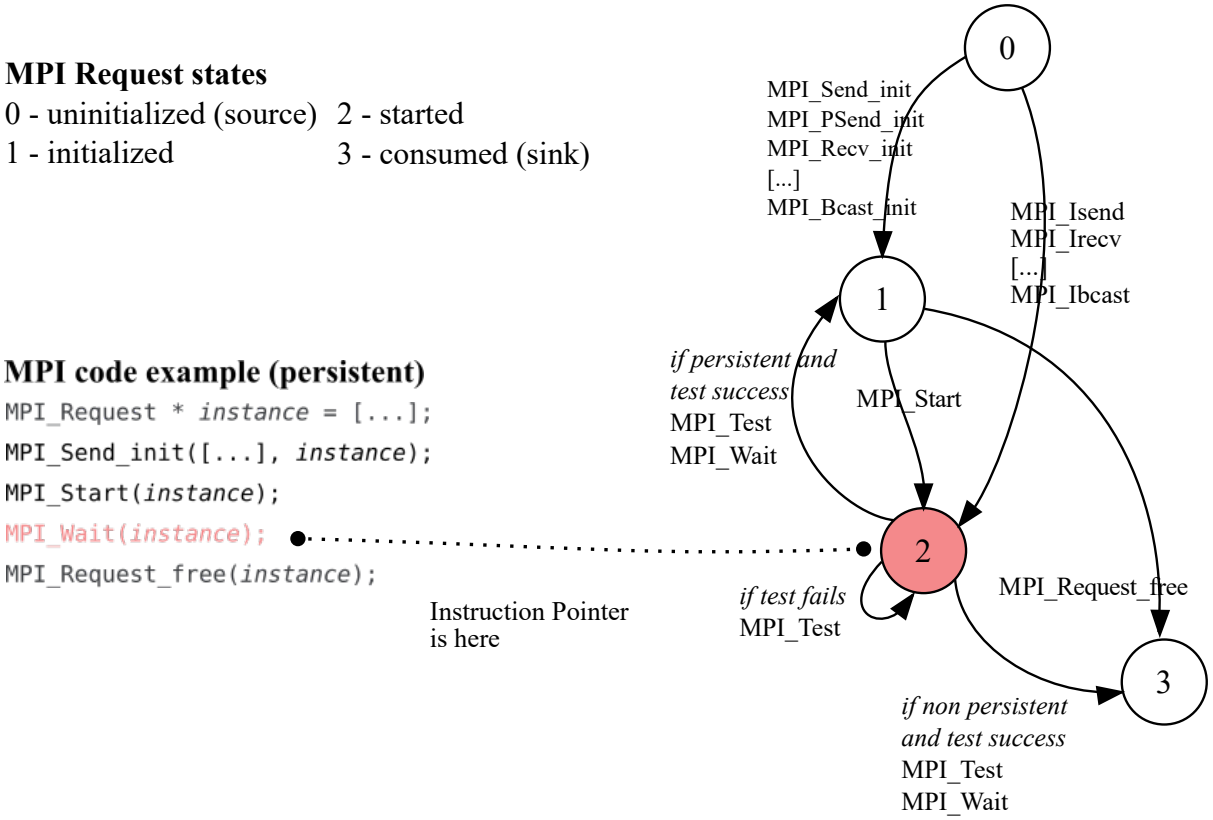


Figure 3.1: MPI Requests as a transition system

The Request API The MPI specifications [48] define an `MPI_Request` transition system, depicted on Fig. 3.1. The colored node shows the left-side instance state after processing instructions line 5. An `MPI_Request` is a transition system instance representing a local handle for a communication that can be:

- *synchronous* if `MPI_Request` instances involved in the same communication must be started before any instance can be consumed, or *asynchronous* otherwise.
- *point-to-point* or *collective* respectively if only two MPI processes are involved in the communication (a receiver and a sender), or if it involves a group of MPI processes. In particular, many collective types had been developed to optimize specific communication pattern encountered in scientific simulation codes. For instance, the `Allreduce` collective defines a global reduction operation (such as a sum, an extremum...) whose result is returned to each member of the group.
- *persistent* or *nonpersistent*. An MPI request can be marked as persistent on initialization using the `MPI_*_init` family routines (* ∈ {Send, Recv, ..., Bcast}) as shown on Fig. 3.1. Otherwise, the request is nonpersistent. The original motivations of persistent requests was to amortize communication initialization overheads by executing only once the transition ⓪ → ① but multiple times ① → ②.
- *partitionned* if the message data may be cut into multiple partite, where each partite can be marked "ready" by the programmer using the `MPI_Pready` routine. As soon as a partite is ready, it may be sent/received independentely to one another. A *nonpartitionned* MPI request can be seen as a 1-partite MPI request (e.g. `MPI_Send`). MPI partitionned requests original motivations were two-fold [71]. The first motivation was similar to persistent requests: overhead reduction by executing ⓪ → ① only once. The second motivation is

early-bird posting, so network transfers can start as soon as the data is locally ready: using MPI partitioned requests, a partite may be sent as soon as its data was written to the send buffer without the need to fill the buffer entirely with every partite of the message. Note that in the current specifications, partitionned requests are necessarily point-to-point and persistent requests.

MPI Requests Progression Transitioning MPI request instances requires some CPU cycles to ensure run the communication protocol logic. For instance, this is the case for routing hierarchical messages or simply emptying the network interface controller memory buffers. *Progressing* MPI requests consist in dedicating CPU cycles to run this logic. Even though it is not clearly defined in the standard specifications yet[1] the community distinguishes two types of communication progression: *strong* and *weak* progressions.

Strong progression can be seen as mechanisms providing *upper bounds on communications time*. For instance, kernel threads preempting the execution dedicated to MPI communication progression had been studied by H. Taboada [72] and F. Reynier [73]; with a study on the impacts of dedicating or ovesubscribing (having multiple kernel threads competing CPU time) cores on computational performances [74]. Other techniques, such as Direct Memory Accesses (DMA) by the NIC or offloading communication progression logic to computing units integrated into the NIC (known as "Smart NIC" [75]), can also be considered as strong progression mechanisms.

Weak progression does not provide bounds on communication time. It often refers to lazy and opportunistic progression mechanisms. Pioman [76] is a pthread-based multi-threaded communication engine whose design considered weak progression on idle periods or any explicit point raised by the program. MPC collaborative polling [77] is another form of weak progression: any thread may progress MPI requests during idle periods, which can improve latency and bandwidth in some cases by dynamically adjusting the number of cores contributing to communication progressions.

In this thesis, we extended MPC collaborative polling so MPI requests can be polled not only during idle periods, but also on specific OpenMP tasking events. This work is further presented in Section **??**.

Implementations Currently, the two main implementations of MPI are Open MPI [78] and MPICH [79]; and their derivatives. Open MPI derivatives include for instance, Bull-MPI for BXI Interconnect, which relies on the Portals4 low-level network API[2]. MPICH derivatives include, for instance, the Intel MPI Library, Cray MPT, or MVAPICH; which standardized their Application Binary Interface (ABI) so the MPI runtime implementation can be switched without the need to recompile the MPI program. Both implementations provide high-level standard specification support and performances on existing interconnection networks. Their working method by publications and open-source implementations lead to a friendly competition in which one can unlikely outperform the other for a long time[3].

A few more MPI implementations non-derivative from Open MPI or MPICH exists, but at much less used in production environment and are mainly used for research purposes. MPC-MPI [11] implements MPI processes as threads. MAD-MPI is built on top of the New-Madeleine [80] communication engine, and was originally motivated to improve non-continuous memory transfers. ExaMPI [81] is "*an experimental MPI implementation designed to simplify learning, modifying and use for middleware research*".

[1] `https://github.com/mpi-forum/mpi-issues/issues/637`

[2] `https://www.sandia.gov/portals/portals-4-0-specification-clone-2/`

[3] `https://stackoverflow.com/questions/2427399/mpich-vs-openmpi` - J. Hammond on differences between Open MPI and MPICH

Conclusion The MPI specifications include many more interfaces for managing inter-MPI process communications on supercomputers, such as: network topology, file I/O, one-sided operations, sessions... As part of this thesis, we primarily focus on the MPI request interface. In Chapter 5, we explore how it can be mixed with the Open Multi-Processing (OpenMP) standard programming model to improve data transfers with early-bird posting, and weak-progress not only on idle periods, but also opportunistically in the MPC-OMP runtime.

3.2 Open Multi-Processing (OpenMP)

OpenMP is a programming model initially designed for intra-node parallel programming. Its specification is a 600+ pages document defining C/C++/Fortran extensions with compiler directives, library interfaces, and environment variables. In addition, some members of the Architecture Review Board (ARB) in charge of maintaining the standard specifications also developed a document with a minimal example for each interface[4]. The two main implementations of the standard are currently LLVM OpenMP [82] and GNU OpenMP [83] implementations, respectively with 587 and 228 commits merged in their production branches between February 01 2022 and March 30 2023. They both extend C/C++ compilers and provide a runtime system.

Other implementations exists but are mostly research projects with few uses in production environment. For instance, MPC-OMP [84], XKaapi [60] or Argobots [85] implements their own OpenMP runtime relying on GCC/LLVM Application Binary Interfaces (ABIs). OmpSs [86] and its compiler Mercurium is a programming model extends the C programming language with directives similarly to OpenMP, and influences the standard specifications.

3.2.1 Before Tasking

The OpenMP 1.0 specifications were originally proposed as a portable programming interface for shared-memory multi-processor in 1997 [46]. Back then, it was proposed as a simpler alternative to MPI, HPF and pthreads for shared memory parallelism, which we judged too low level for scientific applications. For instance, this very first version already included the `parallel do` (Fortran) and the `parallel for` (C) constructs to execute loops iterations in parallel, distributing iterations evenly and continuously between threads. This is also known as *implicit* tasking. Version 2.0 published in 2002 provided minor additions, for instance, with the addition of the `num_threads` clause onto the `parallel` construct, which could only be configured globally through `OMP_NUM_THREADS` variable previously.

3.2.2 After Tasking

Version 3.0 published in 2008 extended specifications with the concept of *explicit* tasks with the `task` construct heavily inspired by Cilk 5 [87]. This was the first step in moving OpenMP to a task-based programming model that led to significant work over the past 15 years.

The original tasking design was motivated to improve the expression of irregular parallelism [88] on compute-node with an increasing number of CPUs. This first version of the OpenMP task specification [89] only provided means to manage independent tasks in a control-flow graph. To ensure the correct order of execution, programmers were given barriers (`taskwait` construct, or implicit region barriers), which suspends the execution of the current task until every previously created children tasks completed. Interfaces for managing dependent tasks were only added in specifications 4.0 (2013) five years after the extension proposal [90] to provide finer control on the order of execution through a dependency graph. For instance in 2014, the Kastors benchmark suite [91] showed that dependent tasking could outperform independent tasking on SparseLU and Strassen benchmarks. Specifications 5.0 in 2018 introduced a few new tasking features, such as

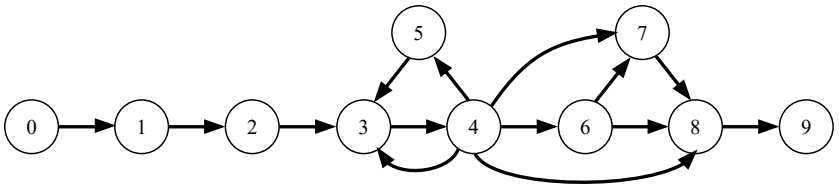
[4] `https://github.com/OpenMP/Examples`

the `taskloop` construct, the `detach` clause to differ completion until an external event is fulfilled, and the capability to perform task-based reductions.

3.2.2.1 Tasks as a Transition System

Task Dependency Node states

0 - uninitialized
1 - not_queuable
2 - queuable
3 - queued
4 - scheduled
5 - suspended
6 - executed
7 - detached
8 - completed
9 - deleted



Transition	Occurs	Concurrency	Note
⓪ → ①	- upon encountering a `task` or a `target nowait` construct (l.9)	1	state ① is internal to the runtime, and protect from being `queued` until every dependences are treated (l.16)
① → ②	- upon encountering a `task` or a `target nowait` construct (l.14)	1	state ② is internal to the runtime, ensuring tasks may not be queued until each dependency had been treated
② → ③	- upon encountering a `task` or a `target nowait` construct (l.15) - after a predecessor completed (l.25)	n	The producer or any consumer thread may transition once each predecessor executed
③ → ④	- on any scheduling point (l.28)	n	The specifications define conditions where the current task may change (*scheduling points*) such as the tasking constructs, barriers ...
④ → ⑤	- upon encountering a `taskwait` construct (l.48) - upon encountering a `task` construct of an undeferred task	1	
④ → ③	- upon encountering a `taskyield` construct	1	
⑤ → ③	- once every children tasks has completed (l.53) - if an undeferred task executed	1 or n	1 or n respectively if the task is tied or untied
④ → ⑥	- after the task structure block executed (l.33)	1	
④ → ⑦	- if the task had been canceled and is detachable	1	
④ → ⑧	- if the task had been canceled and is not detachable	1	
⑥ → ⑦	- after executing a detachable task (l.20)	1	
⑥ → ⑧	- after executing a non-detachable task (l.23)	1	
⑦ → ⑧	- after the event of a detachable task is fulfilled (l.23)	1	
⑧ → ⑨	- once the task is deleted	n	the task is implicitly deleted by the runtime; usually using a reference counter

Figure 3.2: OpenMP Task Transition System in the MPC-OMP runtime

Fig. 3.2 presents the OpenMP task implementation in the MPC-OMP [84] runtime as a transition system, and refers to the simplified implementation depicted on Listing 3.1. Note that as a standard implementation, the LLVM, GCC and even Nanos6 [92] implementations are very close to the MPC-OMP behavior.

MPC-OMP tasks are a 10-state transition system, with concurrency on some transitions such as ② → ③ meaning that multiple threads may try to transition the same task instance in

```
1  void init(task_t * task, int size, void * data, void (*f)(), int flags, int priority) {
2      task->f = f;                          /* Save outlined function pointer */
3      task->data = data;                    /* Copy shared/private data */
4      task->omp_priority = priority;        /* Save task priority */
5      setup(task, flags);                   /* Convert passed flags (untied, final, ...) */
6      Set 'task' not_queuable             /* (0) → (1) */
7  }
8
9  void construct(void * data, int size, void (*f)(), int flags, int priority, depends_t * deps)
10 {
11     task_t * task = (task_t *) malloc(size); /* Set state (0) */
12     init(task, flags, priority);
13     for (int i = 0 ; i < deps.size() ; ++i)  /* Link dependences in the TDG */
14         dependency_insert(task, deps[i]);
15     Set 'task' queuable                     /* (1) → (2) */
16     process(task);
17 }
18
19 void finalize(task_t * task) {
20     if (task->detach_event) {             /* If a 'detach(event)' clause is specified */
21         Set 'task' detached                /* (4),(6) → (7) */
22         wait(task->detach_event);
23     }
24     Set 'task' completed                    /* (4),(6),(7) → (8) */
25     for (task_t succ : task->successors)
26         process(succ);
27 }
28
29 void schedule(void) {
30     task_t * task = next_ready_task();
31     Set 'task' scheduled                    /* (3) → (4) */
32     if ('task' was not cancelled) {
33         task->f(task->data);
34         Set 'task' executed                 /* (4) → (6) */
35     }
36     finalize(task);
37 }
38
39 void process(task_t * task) {
40     if (each 'task' predecessor state ≥ completed) {
41         queue(task);
42         Set 'task' queued                    /* (2) → (3) */
43     }
44     if (task is undeferred)
45         while (state of 'task' is < executed)
46             schedule();
47 }
48
49 void wait(void) {
50     Set 'current_task' suspended             /* (4) → (5) */
51     while ('current_task' has children tasks in state < executed)
52         schedule()
53     queue(task);
54     Set 'current_task' queued                /* (5) → (3) */
55 }
```

Listing 3.1: OpenMP Simplified Tasking Runtime

parallel. The first column of the table is a transition; the second column provides events/interfaces that can lead to transitioning; the third column is the level of concurrency, that is the amount of thread that may concurrently try to transition a task instance; and the last column provides additional information on the transition.

On the simplified runtime implementation, the `init` and `construct` routines create a new task. The `data` parameter represents the task passed `private` variables, but could be more generally assimilated to a "task local storage". The `dependency_insert` routine inserts the task into the dependency graph. Any thread may run the `schedule` routine upon reaching a scheduling point; that is when a new task may be elected to run. After executing the task, the thread runs the `finalize` routine waiting for the associated event if the task had been detached, and then fulfill its dependencies. The `process` routine queue the passed task if its dependences are met, and waits for its execution if it is an undeferred task; that is, with respect to the `if` or `final` clauses, or if the runtime scheduler internally decides to serialize it through task throttling [93] to prevent excessive memory use if enough tasks had already been created. For instance by default, GCC and LLVM respectively limits the number of pending tasks to $64 \times nthreads$ and 256 tasks.

Standard version	Additions
< 3.0 (< 2008)	- `parallel for`, `section`, `single` (implicit task)
3.0 (2008)	- concept of tasks in the execution model - `task` construct (explicit tasks) - `taskwait` construct (explicit synchronization)
3.1 (2011)	- `taskyield` construct (explicit scheduling point) - `final`, `mergeable` and `depend` clause to the `task` construct
4.0 (2013)	- `taskgroup` construct - `cancel` construct with the `taskgroup` clause) - `depend` clause on the `task` construct (implicit synchronizations) - `target` construct (synchronous target task)
4.5 (2016)	- `taskloop` construct - `priority` clause on the `task` construct - `nowait` and `depend` clause on the `target` construct (asynchronous target tasks)
5.0 (2018)	- array shaping and array sections for (`target update` and `depend` clause) - `iterator` and `depobj` on the depend clause - task-based reductions - `taskloop` clause on the `cancel` construct - `depend` clause on the `taskwait` construct - `mutexinoutset` dependency type - `detach` clause on the `task` construct
5.1 (2021)	- `grain_size` and `num_tasks` clause on the `taskloop` construct - `inoutset` dependency type - `nowait` clause on the `taskwait` construct
5.2 (2022)	- `omp_in_explicit_task` API

Figure 3.3: OpenMP Specifications evolutions on tasking concepts

3.2.2.2 Target Tasks

As accelerators (such as GPU) were appearing in supercomputers nodes, the OpenMP specifications provided the `target` interface in its specification 4.0 to enable portable computation offloading [94]. Programmers can define a `target` code region to be executed on a specific *device* connected to the *host* compute node. The memory model assumes that memory space between the host and the device are different: if memory spaces are unified, programmers must explicitly specify the `requires(unified_shared_memory)` clause. The execution model is host-centric:

the host decides explicitly which data to move when to move, and on which device. Initially, the `target` was separated from tasking in the specifications, but version 4.5 (2015) added the `nowait` and `depend` clause so `target` region are considered as dependent tasks, allowing fine synchronizations between the host and the device computation and memory transfers using the task dependency graph.

3.2.2.3 Conclusion

To follow intra-node evolutions, the OpenMP specifications had lived several extensions since its original specifications in 1999. Since the many-cores era in the mid 2000s, specifications had been moving towards a task-based programming model; which task-related evolution is summarized on Fig. 3.3. Still, the specifications original spirit of focusing on intra-node parallelism mostly remains. Therefore, stand-alone OpenMP is not a sufficient solution to efficiently exploit a supercomputer with distributed-memory. Another programming model, such as MPI, is required to perform memory movements over the inter-connexion network, which had lead the HPC community to mix both programming models.

3.3 Hybridizing MPI and OpenMP

Before the many-core era that started in the 2000s, supercomputer compute-node used to be single-core processors with a NIC interconnected in a network. Parallelizing applications using only MPI was sufficient to exploit the machine efficiently [70]. In particular, the Bulk-Synchronous Parallel (BSP) [95] bridging model was widely implemented in application codes, where processors repetitively perform 3-stages super-steps: computation, communication, and synchronization. In the model, computation from super-step $n + 1$ cannot start until the synchronization of every processor of the super-step n is completed. Note that in the original model, each super-step has a fixed time duration; but hardware and software implementations were not rigorously providing such guarantees.

3.3.1 Historical Bulk-Synchronous Parallel Program Structure

As compute-node became themselves parallel architecture with the apparition of many cores, relying only on MPI was no longer sufficient [70]. A mixed-use of standard presents benefits to exploit this new intra-node parallelism using a shared memory programming model such as OpenMP; in particular for applications suffering from load imbalance between MPI processes, memory limitations due to data replication or a restriction on the number of MPI processes combinations. In 2009, M. Tsuji et al. [96] conducted evaluations of the T2K supercomputer reporting results in that direction: mixed versions provide improved performances over MPI-only versions of the evaluated benchmarks on a multi-core multi-socket node cluster, and using one MPI process per socket running a multi-threaded OpenMP runtime. Since then, mixed-use of the standard became usual and found in a wide range of applications, but mostly consists in preserving existing BSP programs parallelizing the computation stage (and sometimes the communication stage) using `# pragma omp parallel for` loops. However, preserving original BSP program structure, communications may not start until its executing thread previously completed its computation stage, limiting the potential of network communication overlap with computation [6]. In addition, the synchronization stage remains which limits load balancing and overlap furthermore.

3.3.2 Task-based Hybridization

On the other hand, mixed use of the standards using a task-based composition provides additional advantages over a BSP use. In particular, it allows the *implicit* overlap of communication with

independent computation [97] as the task dependency graph expressed by the programmer provides the degree of parallelism to the task scheduler. However, it requires porting applications from a BSP to a task-based model; and executing MPI communications within OpenMP tasks most likely leads to the *loss* of an OpenMP worker thread issue.

3.3.2.1 Nesting MPI Communications into OpenMP Tasks

```
1   # pragma omp task depend(out:  x)
2   {
3       [...] /* some work on which depends on the request completion */
4       MPI_Start(&req);
5       [...] /* some independent work */
6       MPI_Wait(&req, &status);
7       [...] /* some work that depends on the request completion */
8   }
9
10  # pragma omp task
11  {
12      [...] /* some independent work */
13  }
```

Listing 3.2: MPI and OpenMP task-based hybridization

Listing 3.2 presents a minimal and motivating example illustrating the loss of an OpenMP worker thread under a task-based hybridization. The objective of this code is to overlap the MPI synchronization line 6, switching the executing thread to any other ready tasks (such as the one line 10). However, such behavior is not granted by standards. In practice, the thread would block into the MPI runtime line 6, leading to the loss of worker thread [98] on the OpenMP runtime; and ultimately to low performances or even deadlocks. This problem motivated the community to propose various solutions.

Solution (1) Test+Yield In [4], programmers replaced the `MPI_Wait` with a loop calling `MPI_Test` and the `taskyield` construct as shown on Listing 3.3. Authors initially assumed the executing thread would schedule any other ready tasks until the request is completed.

```
1   # pragma omp task
2   {
3       while (1)
4       {
5           int completed;
6           MPI_Test(&req, &completed, &status);
7           if (completed) break ;
8           # pragma omp taskyield
9       }
10  }
```

Listing 3.3: Suspending tasks using the taskyield construct

However, the OpenMP standard does not guarantee such behavior, as it only specifies that: "*The taskyield region includes an explicit task scheduling point in the current task region.*". It means the executing thread may not necessarily switch to ready tasks. This motivated J. Schuchart et al. [99] to explore various implementations on the `taskyield` construct for switching tasks. Their `circular-yield` algorithm suspend the current task until every other task executed at least once, enabling the expected behavior with MPI. Their `stack-yield` algorithm executes a new task on the current thread on top of the current one. Regarding OpenMP runtime implementations, GCC and MPC-OMP `taskyield` construct is a `no-op`.

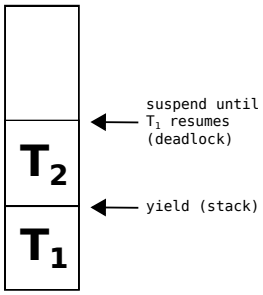


Figure 3.4: Deadlock of stack-yield algorithm

With LLVM, if the task is annotated `untied`, the compiler automatically privatizes variables from the outermost scope to a task local storage. It also rewrite its memory accesses accordingly. During the execution, as long as the instruction pointer remains in the task outermost scope, the task remains *stack-less*. In such case, the entire execution context fits into fixed-size task' local storage. Hence, whenever the executing thread reaches a scheduling point on the outermost scope, it can return to the LLVM OpenMP runtime leaving no footprint on the current thread stack and get re-queued. When the task is re-scheduled, the underlying thread jumps to the scheduling point where the task was suspended. As an explicit scheduling point, the `taskyield` construct follows this rule if appearing in the outermost scope of an `untied` task; else if the scheduling point appears in a nested function call, or if the task is `tied`, LLVM uses a `stack-yield` algorithm. Hence, LLVM mixes J. Schuchart and al. approaches but presents limitations:

- The `stack-yield` algorithm triggering by default can lead to the deadlock of valid OpenMP programs. Fig. 3.4 is a minimal example deadlock scenario: T_1 is scheduled, yields to T_2, and T_2 yields until T_1 completed. T_1 will never be able to resume as it is blocked T_2 on the thread stack, leading to a deadlock. The OpenMP program of this scenario is provided in Annex 8.4.

- A `circular-yield` alike algorithm triggers only on `untied` tasks, and if scheduling points appear on the outermost scope of the task. However, programmers *cannot* use Variable Length Array (VLA) or the Linux `alloca` function, as the thread stack may be rewritten between scheduling points. Doing such currently causes the compiler to crash[5].

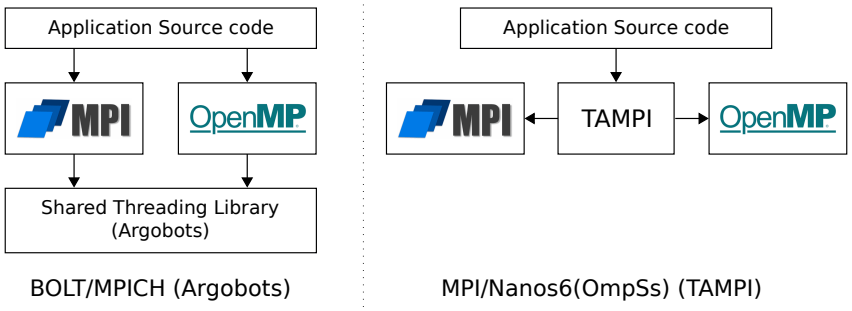


Figure 3.5: MPI and OpenMP interoperability

Solution (2) Runtime Interoperability A second solution consisted in preserving existing code structure and having runtimes implicitly managing task suspension.

MPICH+ULT [100] (2015) proposed to automate threads suspension by using Argobots as a threading library for MPICH. In 2019, they implemented an OpenMP runtime (BOLT [101]) that also relies on the Argobots threading library. Their objectives were to automate interoperability between MPI and OpenMP through the common Argobots threading library. With such approach, as opposed to TAMPI, user code may remain unchanged. However, BOLT only provided Argobots support for threads on `parallel` region, and have not yet investigated explicit `task` suspension.

The Task-Aware MPI (TAMPI) [6] library was proposed as an extra layer above MPI, to suspend tasks that would block in the MPI runtime otherwise. The authors provided an implementation for the Nanos6 tasking runtime of the OmpSs standard. Using this library,

[5] `https://github.com/llvm/llvm-project/issues/61499`

programmers must replace MPI blocking calls with their TAMPI equivalent (for instance, `MPI_Wait` by `TAMPI_Wait`).

Both approaches are summarized on Fig. 3.5. Nevertheless, no standard solutions have been adopted in the standard OpenMP or MPI specifications when executing blocking MPI operations in a task-based environment, so it remains implementers and programmers responsibility to ensure there is no loss of threads.

Solution (3) Task Detach A third solution came with the task' `detach` clause introduced in OpenMP 5.0 specifications (Nov. 2018). Set on a task construct, the clause differs its completion until an external event is fulfilled. This would be used in codes as depicted in Listing 3.4 which we retrieved from [6, 102] as follows: Using the `task detach(event)` construct, application

```
1   omp_event_handle_t ev_handle;
2   # pragma omp task detach(ev_handle) depend(out:  y)
3   {
4       [...] /* some work on which depends on the request completion */
5       MPI_Start(&req);
6       MPIX_Detach(&req, omp_fulfill_event, ev_handle);
7       [...] /* some independent work */
8   }
9
10  # pragma omp task depend(in:  y) depend(out:  x)
11  {
12      [...] /* some work that depends on the request completion */
13  }
14
15  # pragma omp task depend(out:  x)
16  {
17      [...] /* some independent work */
18  }
```

Listing 3.4: MPI and OpenMP task-based using the detach clause

programmers must express two dependent tasks for managing asynchronous operations: the launch (line 2) and its continuation after completion (line 8). `MPIX_Detach` (line 6) registers the `omp_fulfill_event(ev_handle)` callback on the MPI request completion, which raises an *allow-completion* event to the OpenMP runtime.

This new clause also impacted other asynchronous programming models. For instance, proposals were made to the MPI specifications to register a callback on request completion [102, 103] which are currently being standardized [6][7]. As opposed to the solution (1), this clause provides a portable solution for synchronization overlapping with independent tasks. Nevertheless, as opposed to (2), it increases programming costs for users, moving interoperability responsibility from runtime systems to user codes. Still, this has become the standard way of hybridizing asynchronous programming models using task-based OpenMP.

What about Requests Progression ? All these solutions enable the suspension of tasks waiting on an MPI request completion. It can therefore overlap the MPI request completion time with other independent work through the OpenMP task scheduler. Yet remains questions on the responsibility of progressing communication (*i.e.* such as copying memory buffers, forwarding messages, computing collectives algorithm...) and raising the completion callback. In the `MPI_Detach` proposal [102], authors provided an implementation where a single kernel (p)thread progresses every MPI requests and raises the `omp_fulfill_event` callback on completion. This

[6] `https://github.com/mpiwg-hybrid/hybrid-issues/issues/6`
[7] `https://github.com/devreal/ompi/tree/mpi-continue-master`

strong progress as the kernel preemption on cores ensures that the MPI requests will be granted CPU time to progress regularly, bounding communications during in time. Other approach relying on *weak* progress such as MPC collaborative polling [77] during idle periods is also a solution to progressing MPI requests and raising the OpenMP callback on completion.

3.4 Multi-Processor Computing (MPC)

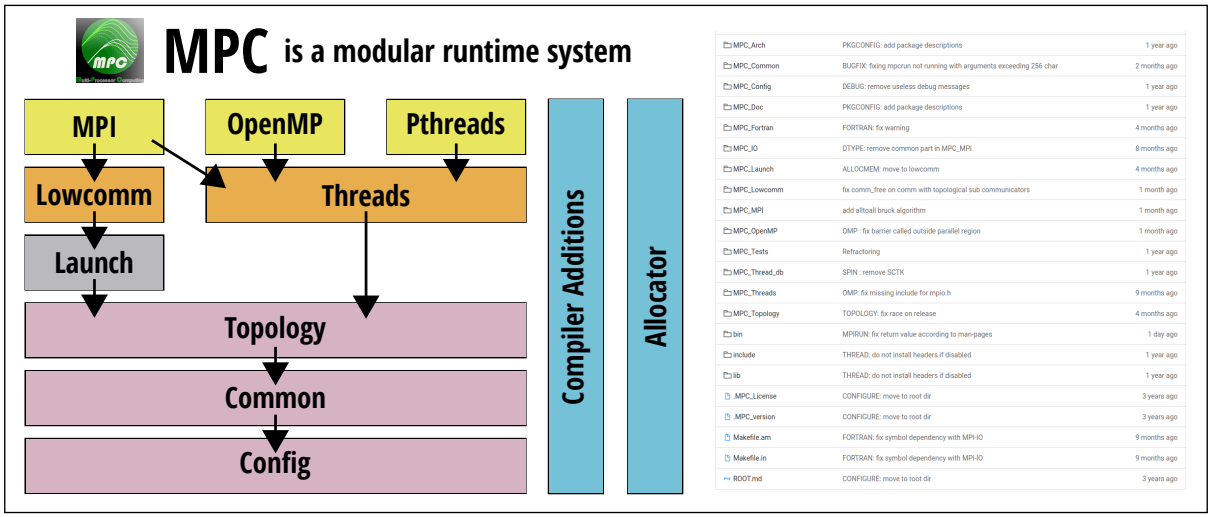


Figure 3.6: MPC layers and source tree overview

The Multi-Processor Computing (MPC) is an implemention of MPI, OpenMP, and pthreads specifications, distributed as a Free Open Source Software (FOSS) [8]. It is an active research project at the CEA. Fig. 3.6 depicts the current software architecture as a modular parallel runtime system.

A Thread-based MPI Implementation MPC initial design in 2008 consisted in a unified implementation of POSIX Threads (pthreads) and MPI. pthreads and MPI processes are virtualized with user-level threads known as "MPC threads" [11]. The shared "Thread" layer of the MPC software architecture is responsible of scheduling each MPC threads on *virtual processors* (VPs) implemented as Linux kernel threads. While existing MPI runtimes used to implement MPI processes as Linux processes, MPC-MPI processes running on the same compute-node may run as MPC threads in the same virtual memory address space. This created issues for supporting existing codes, in which developers assumed that an MPI processes are always a POSIX process with their own virtual address space. For instance, *global variables* were assumed private per MPI processes unlike thread-based MPC-MPI processes. Therefore, compilers and linkers had been extended to support an extra layer of Thread Local Storage (TLS) which stores the MPI process data [104]. The MPC compiler privatizes global variables into this TLS so that each MPC-MPI process has its own copy, supporting previous code assumptions.

The MPC OpenMP Implementation In 2010, MPC was extended with an OpenMP runtime supporting both GCC and Clang ABIs [84]. OpenMP threads were implemented as MPC threads scheduled by the shared "Thread" layer like any other threads. In the "OpenMP" layer, A. Mahéo et al. [105] organized MPC-OMP threads in a tree based on hardware topology primarily motivated to accelerate thread synchronizations and activations. Fig. 3.7 illustrates such representation. In this example, the last level 2 represents cores (threads), level 1 represents

[8] `https://github.com/cea-hpc/mpc`

the NUMA domain shared by cores, and level 0 is the compute node. In their conclusion, A. Mahéo et al. suggested reusing this data structure for a topology-aware task scheduling strategy; so that threads favor stealing on the closest threads queue in the hardware. They since then implemented into the MPC-OMP runtime and evaluated as part of his Ph.D. manuscript [106].

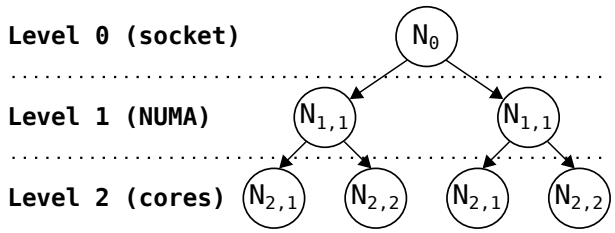


Figure 3.7: Topological Representation of Threads

MPC in this thesis As part of this thesis, we mostly restricted our use of the MPC runtime to the OpenMP layer. Threads are always bound 1:1 onto virtual processors, themselves bound 1:1 to physical cores, mostly ignoring low-level MPC threads details to focus on OpenMP aspects. One source of motivation to that decision was to take advantage of previous work on the MPC-OMP work-stealing task scheduler by A. Mahéo, which we heavily re-used throughout this thesis.

3.5 Conclusion

Since the many-core era, mixing OpenMP and MPI showed to be crucial to efficiently exploit supercomputers. Original mixed-use were partly motivated by memory overheads of the MPI-only approach; on which OpenMP provide a response as a shared-memory programming model. Since the many-core era, OpenMP had been moving to a task-based programming model that could offers new benefits to the mixed-use with MPI. For instance, the overlap of MPI communications with OpenMP tasks through a task dependency graph scheduling was recently studied (2015-2019) [6]. However, interoperability issues makes it hardly achievable in practice, with poor performance portability of existing solutions over environments of execution. In Section 5.1, we introduce mechanisms in the MPC-OMP runtime for suspending/resuming tasks in the presence of MPI synchronizations (implementing effective `taskyield`) towards the automation and portability of runtime interactions. Mixed-use of the programming models using tasks presents another important benefit: the early-bird of MPI communications by OpenMP dependent task scheduling. In Section 5.2, we present how we extended the MPC-OMP tasks with *priorities* to guide the scheduler to electing tasks performing communications at the earliest.

In order to evaluate these extensions on real-world applications, Chapter 6 presents our porting experience of two mainstream HPC applications. During our journey, we discuss the three profiling/programming/performances difficulties introduced earlier. For instance, we propose standard and MPC-OMP runtime extension on task in Chapter 5 for task suspension/resumption, and standard extensions in Section 6.1.1 to ease the expression of irregular dependencies. Evaluations suggested the task dependency graph creation can be a performance bottleneck in current OpenMP implementations, that lead us yet again to extend the standard and MPC-OMP task transition system with *persistence* in Section 6.2.3.2.

But first, the next Chapter 4 introduces the profiler we developed to overcome the current lack of tools for dependent task-based hybridization; that we extensively used in almost all of the experiments of this thesis.

Part II

Contributions

Chapter 4

Measuring Performances of Task-based Applications

Contents

This chapter present our contributions on the performance *profiling* difficulty of MPI and task-based OpenMP applications. We respond to the lack of tool dedicated to dependent task-based programming proposing an hybrid profiler. First, we present a modeling of hybrid applications to a unified task graph scheduling problem, in order to formally define performance metrics in the context of dependent task-based programming. Then, we implement a profiler and analysis to synthesize metrics from actual executions.

4.1 Defining Metrics

Applications are often modelled through a graph where nodes represent sequential sets of instructions (tasks) and edge their precedence constraints [107, 108, 109]. Task Graph *Scheduling* is then studied as the assignation of tasks to their starting on a processing unit with solutions as couples (σ, a) mapping each node v to its starting time $\sigma(v)$ on a processor $a(v)$.

4.1.1 Task Graph Scheduling Problem and Observation

Classical modelings assume that each node and edge completion time is fixed and independent on the schedule itself. However multiple phenomena can cause tasks completion time to vary at execution. Hierarchical memory accesses [9], process stalls from cache misses [110], memory contention [111], network contention [10], or even Dynamic Voltage and Frequency Scaling (DVFS or CPU throttling) [112] are examples of what can impact the duration of the exact same workload depending on the *scheduling itself*. Therefore in our modeling, nodes, and edges

completion time are part of the scheduling solutions themselves, named below as *observations* to be constructed by profiling executions.

We define a *Problem* as a processor set $\mathcal{P} = \{1, 2, ..., p\}$ with $p \in \mathbb{N}^*$ the number of processors and an application as a directed acyclic graph (V, E) with V the nodes representing tasks and E the edges representing tasks precedence constraints. An *Observation* is a triplet (σ, a, d) with

- $\sigma : V \mapsto \mathbb{R}^+$ mapping tasks to their starting time,
- $a : V \mapsto \mathcal{P}$ mapping tasks to processors,
- $d = (d_t : V \mapsto \mathbb{R}\ ,\ d_c : E \mapsto \mathbb{R})$ mapping nodes and edges respectively to work and communication time,

veryfing the *precedence* (4.1) and the *processor* (4.2) constraints:

$$\forall e = (u, v) \in E,$$
$$\sigma(u) + d_t(u) + d_c(e) \leq \sigma(v) \tag{4.1}$$
$$a(u) = a(v) \Rightarrow \begin{cases} & \sigma(u) + d_t(u) \leq \sigma(v) \\ \textbf{or} & \sigma(v) + d_t(v) \leq \sigma(u) \end{cases} \tag{4.2}$$

4.1.2 Metrics Definition

Given a problem $(\mathcal{P}, V, E)$ and an observation $S = (\sigma, a, d)$, we define multiple to assess the performances.

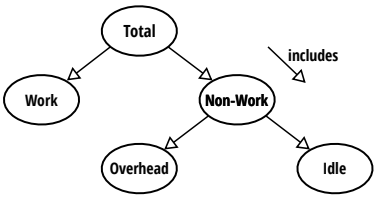


Figure 4.1: Time Breakdown

Time Breakdown In [113], the authors proposed the time breakdown depicted in Fig. 4.1 to assess multithreaded application performances. It allows performance characterization of a multithreaded execution to balance the parallelism (idleness) with management costs (overheads). They illustrated this time breakdown on OpenMP before introducing *dependent* tasking [114] in 2009. Hence, we adapted the time breakdown for dependent tasking while preserving the sense of these metrics. We define the *total* time as the duration from the first task schedule to completing the last task on any threads. The *work* is the time spent within an explicit task program, the *overhead* is the time spent outside of any explicit task program while there are tasks ready, and the *idleness* is the time spent outside of any task program while there are no tasks ready.

Work Time Inflation The time breakdown provides an overview of the parallelism but lacks details on the work time performances. S. L. Oliver and al [9] introduced the *work time inflation* metric defined as *"the additional time spent by threads in a multithreaded computation beyond the time required to perform the same work sequentially."*. They illustrated that work time can significantly inflate in the presence of non-uniform memory accesses. However, work time inflation can be seen more generally as a matter of *task scheduling*, instead of a matter of the number of threads. For instance, depth-first scheduling can improve cache reuse [8] and reduce the work time, independently of the number of threads. Therefore, we propose a different definition of work time inflation: instead of an *"additional time"* (in seconds), we define the *work time inflation* i_t as a ratio. (no units) between two observations $S^{ref} = (\_, \_, d^{ref})$ and $S = (\_, \_, d)$.

$$i_t(u) = \frac{d_t(u)}{d_t^{ref}(u)} \quad \textbf{and} \quad i_t(V) = \frac{\sum\limits_{u \in V} d_t(u)}{\sum\limits_{u \in V} d_t^{ref}(u)}$$

$i_t(u)$ is the work time inflation of the task u for the observation S, using the u work time on the observation S_ref as a reference. $i_t(V)$ is the global work time inflation for the observation S using the observation S_ref as a reference. In practice, we usually choose S^{ref} the observation with the least work time.

Communication Time and Inflation P. Swartvagher and al. [111] studied the impact of concurrent communication and computation on performances. In particular, they have shown that communication time can increase in case of memory contention where CPUs and NICs concurrently access the DRAM. In task-based applications, similarly to work time inflation, communication time degradation can be seen as a scheduling matter. We define *communication time inflation* metric to quantify degradation due to scheduling decisions, between two observations observation S^{ref} and S as:

$$i_t(e) = \frac{d_c(e)}{d_c^{ref}(e)} \quad \textbf{and} \quad i_c(E) = \frac{\sum\limits_{e \in E} d_c(e)}{\sum\limits_{e \in E} d_c^{ref}(e)}$$

In practice, we usually choose S^{ref} the observation with the least communication time.

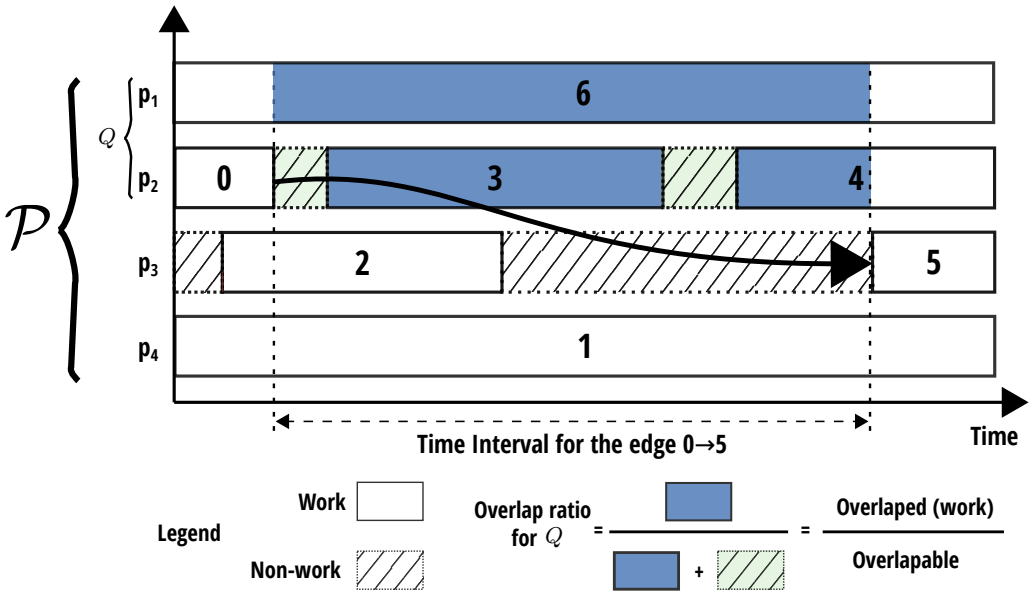


Figure 4.2: Overlaped work on a communication edge

Communication Overlap with Computation The overlap of communication with computation is the capacity of CPUs to perform computation while data transfers occur asynchronously onto the network. Dependent task parallelism enables automatic communication overlap with computation having cores switching to ready tasks during synchronizations. Such an approach was used with SmpSs in [97, 115], but the overlap itself was not explicitly quantified, making it hard to ensure that overlap is the reason for observed performance gain (and not other phenomena such as work/communication time inflation).

Garcia et al. [3] proposed multiple visualization panels for task-based applications to understand performances from a given task scheduler. In particular, one panel is the network activity over time and another the cores activity over time. Placed on top of each other, this visualization gives hints on the communication overlap with computation. However, this approach is mainly visual, which depends on the analyst's subjectivity, in particular with the presence of visual artifacts.

Richard et al. [116] proposed to report a global ratio that quantifies the overlap. Their approach only assumes that MPI operations are serialized on a single thread and does not permit to report of any information on time sub-interval for finer analysis than ours. Reynier et al. [117] proposed an *overhead ratio* metric and benchmarks to quantify computational/communication overheads when overlapping MPI collectives that are close to our *communication inflation* metric. Just like F. Richard, quantification only considers the activity on a single thread.

We propose a task-based definition of the *overlap ratio* from a given observation to quantify the amount of work processed on a *set of processing units* in parallel with communications on any time window, allowing fine analysis of multi-threaded and multi-phase applications.
Let $Run : P(\mathcal{P}) \times \mathbb{I}(\mathbb{R}) \mapsto P(V)$ mapping the set of tasks running on a set of processors on a time interval. It is defined as:

$$\forall Q \in P(\mathcal{P}),\ \forall I \in \mathbb{I}(\mathbb{R}),\ Run(Q, I) = \{u \in V \mid a(u) \in Q \quad \textbf{and} \quad I \cap [\sigma(u)\ ;\ \sigma(u) + d_t(u)] \neq \varnothing \quad \}$$

The *overlapped work* on the set of processor Q and time interval $[a, b]$ is:

$$\begin{aligned} o_v : (Q, [a, b]) \quad &\mapsto \sum_{u \in Run(Q,I)} min(b, \sigma(u) + d_t(u)) - max(a, \sigma(u)) \\ P(\mathcal{P}) \times I \quad &\mapsto \mathbb{R} \end{aligned}$$

and inferred on edges execution time interval as

$$\begin{aligned} o_v : (Q, e = (u, \_)) \quad &\mapsto o_v(Q, [\sigma(u) + d_t(u)\ ;\ \sigma(u) + d_t(u) + d_c(e)]) \\ P(\mathcal{P}) \times E \quad &\mapsto \mathbb{R} \end{aligned}$$

We define $\hat{E}(Q)$ the set of outgoing edges from a given processor partite $Q \in P(\mathcal{P})$ as

$$\begin{aligned} \hat{E} : Q \quad &\mapsto \{e = (u, v) \in E \mid a(u) \in Q \quad \textbf{and} \quad a(v) \notin Q\} \\ P(\mathcal{P}) \quad &\mapsto P(E) \end{aligned}$$

The *overlap ratio* reduces the overlapped work on every outgoing edges of the processor partite Q

$$r_{overlap} : Q \mapsto \frac{\sum\limits_{e \in \hat{E}(Q)} o_v(Q, e)}{\mid Q \mid \sum\limits_{e \in \hat{E}(Q)} d_c(e)}$$

In practice with MPI, the overlap ratio. is measured using MPI processes as partites Q. Fig. 4.2 illustrates the overlap ratio. definition on the processor partite $Q = \{p_1, p_2\}$ for a graph $G = (V, E)$ with $V = \{0, 1, 2, 3, 4, 5, 6\}$, $E = \hat{E}(Q) = \{(0, 5)\}$. This definition is the multi-threaded generalization of the usual single-thread overlap metrics found in the literature: $\mid Q \mid \sum\limits_{e \in \hat{E}(Q)} d_c(e)$ is an ideal overlapable work time on the multi-threaded MPI process Q during time intervals progressing communications e, and $\sum\limits_{e \in \hat{E}(Q)} o_v(Q, e)$ is the work performed in parallel on any threads.

4.1.3 MPI+OpenMP(tasks): A Unified Modeling

In order to fall back to a task graph scheduling problem, observations, and defined metrics, we model distributed MPI+OpenMP(tasks) applications to a single unified graph: the Global Task Dependency Graph (GTDG).

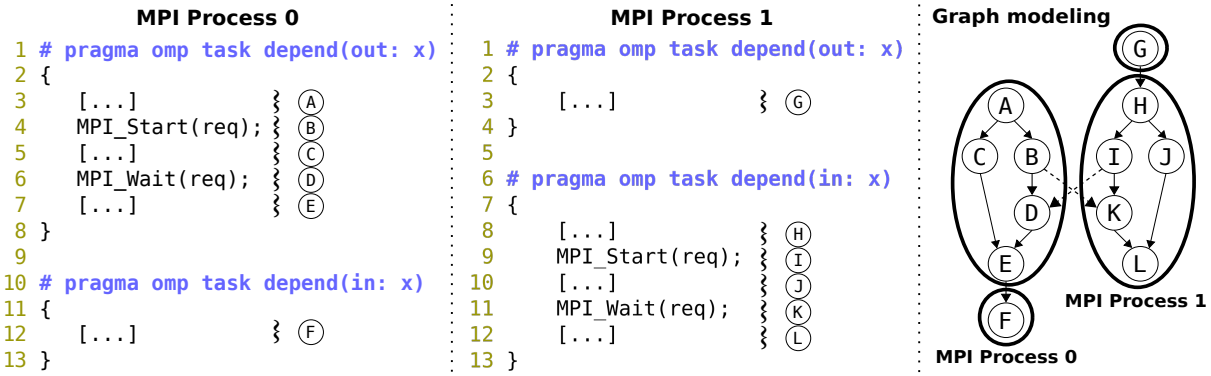


Figure 4.3: From Hybrid Code to a Global Task Dependency Graph

From Hybrid Code to a Global Task Dependency Graph The modeling relies on OpenMP TDG presented in 2.3.1, and MPI requests transition system presented in 3.1. It consists in building a GTDG by refining the OpenMP TDG of each MPI process and interconnecting them to one another. It is created as follows:

- (a) Building the OpenMP TDG of each MPI process,
- (b) Dividing OpenMP task nodes into sub-nodes: each MPI request state transition becomes a sub-node, and sequential instructions in-between MPI calls are grouped into sub-nodes,
- (c) Adding intra-process edges between (b) sub-nodes to preserve both the parallelism expressed through MPI and the sequential order of execution,
- (d) Adding inter-process edges between MPI requests states transition nodes so that transitions to state (3) for *collective*, *receive*, and *synchronous send* operations depends on transitions to state (2) nodes of other processes involved in the communication.

Fig. 4.3 illustrates this algorithm. Bold elliptic nodes correspond to OpenMP tasks on step (a). Nodes $\{B, D, I, K\}$ (MPI request state) and $\{A, C, E, F, G, H, J, L\}$ (sequential instructions in-between) correspond to the OpenMP-task division on step (b).
Paths $\{(A, B, D, E), (A, C, E), (H, I, K, L), (H, J, L)\}$ correspond to step (c) intra-process edges. Dotted edges $\{(B, K), (I, D)\}$ correspond to inter-process edges of step (d).

Note that step (d) must respect both the order of execution and the parallelism specified by MPI. For instance, it models the implicit barrier induced on each collective communication but also models that an `MPI_Wait` on an asynchronous send operation could return before the paired-receive is posted remotely. It also models the explicit overlap parallelism expressed between a started/completed transition state, as shown with independent nodes B and C.

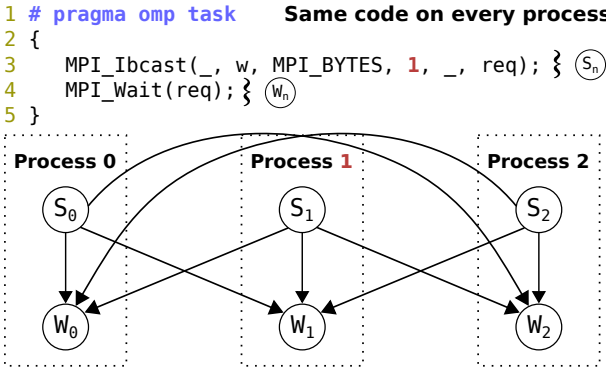


Figure 4.4: MPI Collective Modeling

Fig. 4.4 illustrates another example of an MPI Broadcast collective, with nodes $\{S_i, W_i\}$ corresponding to the initialization and completion state of the MPI request.

4.2 Profiling MPI+OpenMP(tasks) Task-based Execution

Both OpenMP and MPI specifications come with profiling interfaces (OMPT and PMPI) which can allow to build such modelling, and ultimately the defined metrics. The two profiling interfaces are based on callbacks: PMPI overloads every MPI library call while OMPT raises callbacks on specific events. Tools like HPCToolkit [118] or TAU [119] exists and take use of these interfaces to provide information on each programming model. However, they do not compute the time breakdown and the communication overlap we previously defined; as each programming model is profiled independently. Hence, we made a unified profiler using both interfaces. First, it records events during a program execution, and saves them to trace files on termination. After termination (or *post-mortem*), events are replayed sequentially to model the distributed execution to a single task graph scheduling problem $(\mathcal{P}, V, E)$ and an observation (σ, a, d), to synthesize previously defined metrics.

4.2.1 Hybrid Tracing: Recording PMPI and OMPT Events

For our concerns, we profile the states of MPI requests (see 3.1) and task-based OMPT events (implicit task launch/completion, task creation, task schedules, and task dependency completion).

MPC extensions In addition, MPC-OMP provides extra profiling capabilities. A *label*, a *color*, *internal runtime flags*, and up to 4 hardware counters are attached to records. Hardware counters are retrieved using the Performance Application Programming Interface (PAPI) [120] per thread, and on each task schedule. Hence, hardware events can be counted at a task-level precision, in between two task schedule. It enables portable hardware event counters measurements on the *task-level*, as opposed to tools like `perf` or `likwid` which only provide hardware counters on overall execution. The task-level precision enables fine performance characterization on the work, removing idleness/overhead noise.

Moreover, an additional non-standard event has been added: the *task deletion*. The task deletion event is raised when the runtime releases a task that can occur asynchronously any time of the execution. This event helped runtime debugging to track the very asynchronous tasks' lifecycle and ensure no tasks were leaking into memory.

Timing Events Each event is timed using `omp_get_wtime` routine under a microsecond precision and sharing clock time reference between cores of the same compute node. It means that we only have a precisely timed observation per MPI process partite, with no clock synchronizations mechanisms between MPI processes.

From Events to Trace Files From raised events, our profiler generates machine-code *records* storing events information. During the execution, all records are enqueued per core to a pre-allocated memory region and flushed to disk on termination to limit the profiler impact onto the scheduling itself. Profiled execution presented in this manuscript has shown from 0% to 5% slowdown enabling the profiler.

4.2.1.1 From Events to an Observation

An observation is made of four functions $(\sigma, a, d = (d_t, d_c))$, which are to be defined for each graph node from a traced execution. Given a node u, its schedule time $\sigma(u)$ is an OpenMP task schedule event or a PMPI callback on the transition state associated to the node. Its processor

$a(u)$ is set from its OpenMP thread and its MPI rank in the `MPI_COMM_WORLD` communicator. Its duration $d_t(u)$ corresponds to the time taken to execute instructions associated with the node but for MPI nodes $d_t(u) = 0$. Instead, MPI nodes' weights are moved to edges: given an inter-process edge e, $d_c(e)$ is set to the duration of the associated MPI transition state. Intra-process edges duration is set to 0.

4.2.2 Analyzing Traces

In order to synthetize metrics previously defined, we made a framework to analyze execution traces that consists of a *virtual scheduler* and *passes*.

4.2.2.1 Virtual Scheduler

The virtual scheduler reads trace files and replay recorded events sequentially. It provides callbacks and data structures for tracking each OpenMP runtime scheduler event, such as a list of ready tasks, a list of blocked tasks, but also MPI communication state transitions.

4.2.2.2 Analysis Passes

Analysis passes implement the scheduler emulator callbacks; they are called sequentially on the event's callback they implement. Passes can depend on other passes, and the dependencies are automatically resolved by sorting their sequential order of execution (an error is raised in case of a dependency cycle).

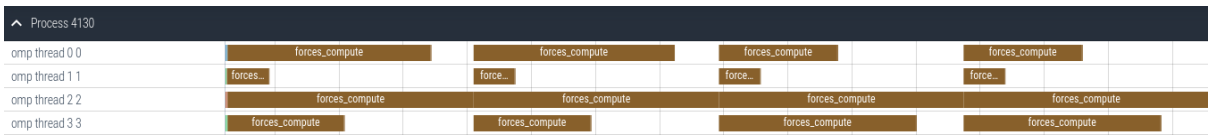


Figure 4.5: Gantt chart of a nbody simulation, showing important load imbalance

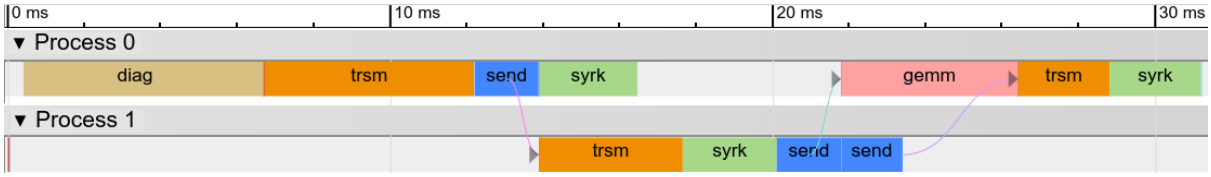


Figure 4.6: Gantt chart of a Cholesky factorization, arrows represent MPI point-to-point communications

List of Analysis Passes Multiple analysis passes have been implemented to build the metrics defined previously. Their source code is available online [1]. Here is the exhaustive list:

- the *Global Task Dependency Graph (GTDG)* builds the unified graph modeling. It provides a graph data structure and synthesizes the observation to be exploited by other passes.
- the *GTDG - Critical* compute the critical path from the GTDG. It implements the shortest path algorithm `24.5 DAG-SHORTEST-PATHS` in [121], by setting edges (u, v) weight to $-d_t(u)$. This pass computes the critical path time, the average parallelism, and execution time bounds proposed by Cilk [87].

[1] `https://github.com/cea-hpc/mpc/tree/cea/2023/icpp-interop-tasks/src/MPC_OpenMP/tools/python`

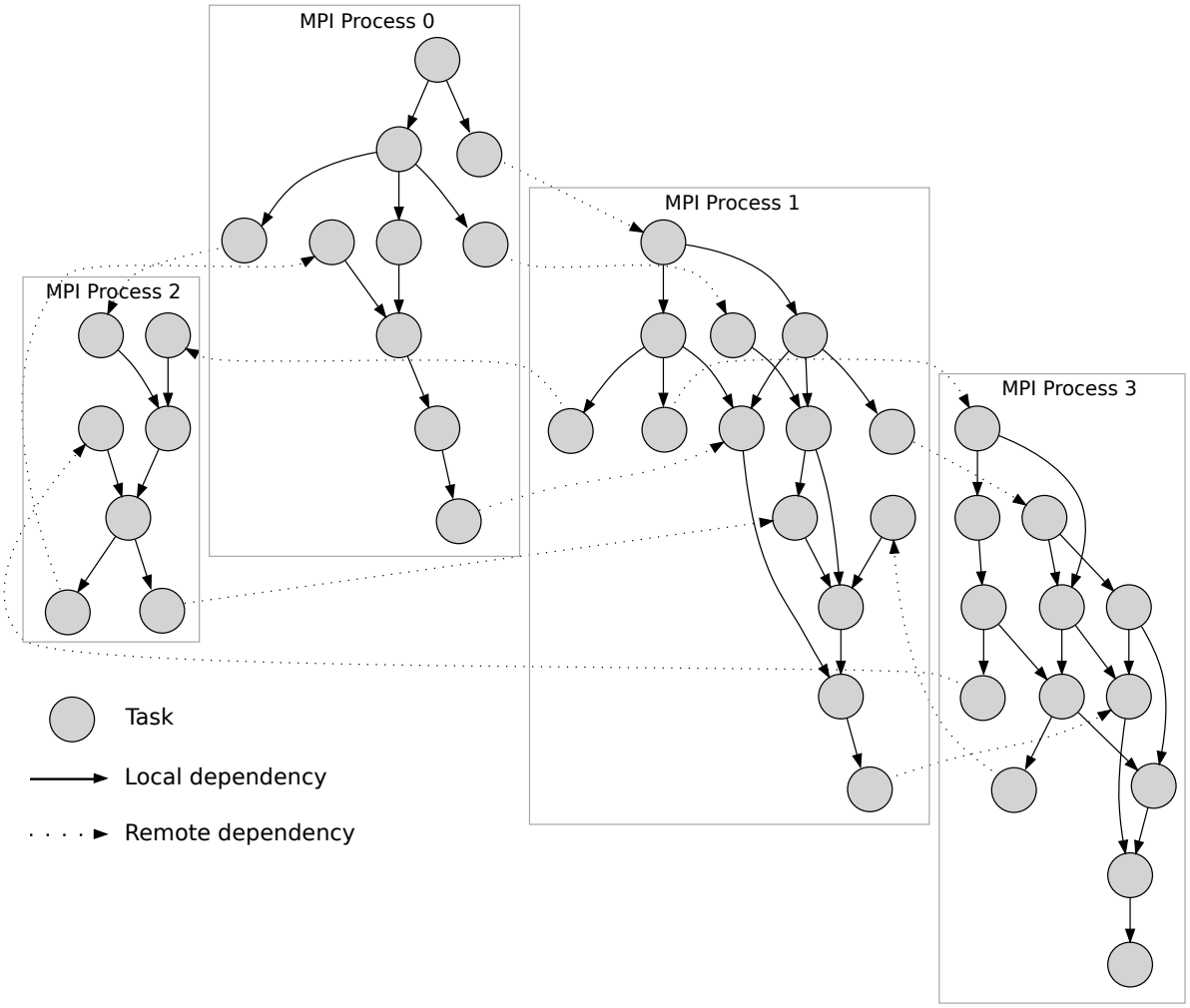


Figure 4.7: Unified Modeling generated by the GTDG analysis pass

- the *GTDG - Dot* exports the TDG to a *.dot* file to be visualized with Graphviz for instance. Fig. 4.7 depicts the modeling for the Cholesky matrix decomposition presented in [99] distributed on 4 MPI processes.
- the *GTDG - Metrics* exports general graph metrics: the number of nodes, edges, and minimum/average/median/maximum degrees.
- the *Time Breakdown* constructs the time breakdown metrics depicted on Fig. 4.1. Work time is the sum of each node duration. Non-work time is split into overheads and idleness by checking the scheduler emulator lists at each raised event.
- the *Gantt* pass export the scheduling observation to a Chrome Trace Event[2] file format, to be imported by a viewer such as Catapult[3] or Perfetto[4]. The y-axis represents the MPI processes and the OpenMP threads, and the x-axis the time. Boxes represent work time (tasks), while the blank spaces represent non-work time. Fig. 4.5 depicts the visualization of a task-based nbody application, in which we can observe load imbalance between tasks leading to important idleness. Fig. 4.6 depicts the Gantt chart of a Cholesky factorization

[2] `https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview`
[3] `https://chromium.googlesource.com/catapult`
[4] `https://ui.perfetto.dev`

on two MPI processes, and arrows illustrate the tasks performing point-to-point MPI communications. Note that times are only coherent for cores using the same time reference.

- the *Communication - Duration* computes the duration of each MPI communications.
- the *Communication - Overlap* computes the overlap ratio.
- the *PAPI* aggregates hardware event counter per node for each schedule event. As opposed to tools like perf or likwid, which only provide hardware counters for the entire Linux process execution, this pass allows the extraction of hardware event counters occurring within explicit tasks for finer performance analysis.

4.2.3 Repos and Documentation

The profiler was built directly into the MPC-OMP runtime for historical reasons (the runtime was not supporting the OMPT interface at the start of the thesis). The MPI profiler is a dynamic library interacting with the MPC-OMP runtime profiler. Post-mortem analyses are built in Python and read from profilers' traces. Source are available online at https://mpc.hpcframework.com or https://github.com/cea-hpc/mpc as free-software under the CeCILL-C[5] license.

4.3 Related Works

Amdahl's Law and Average Parallelism Amdahl's Law provides the theoretical maximum speedup obtainable by increasing the number of processors for a given work time $w = w_s + w_p$ with w_s the in-compressible sequential time, and p embarrassingly parallel time (*i.e.*that could be parallelized on any additional processing units). The law can hardly be applied in practical task-based programming, as there is rarely such thing as pure sequential or embarrassingly parallel time: instead, tasks are interdependent and the amount of parallelism available varies over the execution. Therefore, Cilk' *average parallelism* [122] defined as $\hat{p} = \frac{W_t}{T_\infty}$ is more suitable to determine how the execution time is sensitive the number of processors. The work/idle/overhead time breakdown [113] we adapted to task-based executions is also well-suited for determining execution time sensibility to the number of processors: important idleness may be linked to a lack of parallelism.

OpenMP Tools ScalOMP [123] is an OMPT plugin that can provide hints on loop parallelization and scalability performances over the number of threads. However, it is restricted to loop constructs and does not consider dependent tasking as we do.

Tikki [124] is an OMPT-based plugin that can profile task-based OpenMP applications. It attaches hardware event counters to tasks using PAPI, enabling performance characterization at the task level. Our profiler re-implemented the same capability.

OMPTracing [125] is another OMPT plugin that can profile OpenMP applications. From an execution, OMPTracing generates a Gantt chart and a task dependency graph from a single process OpenMP application. Our approach differs on two points. First, we not only consider task-based OpenMP information but also MPI distribution and build a global task dependency graph partitionned between MPI processes. OMPTracing does not provide such a global view and restricts its analysis to a single process. Secondly, OMPTracing generates its analysis at run-time, while our profiler only records execution information to be analyzed post-mortem. This gives us more flexibility for analysis; for instance, we could synthesize metrics such as the time breakdown or critical path that is hardly doable at run-time. Note that OMPTracing [125] was published in 2020, which is concurrent to my thesis and my work on the MPC-OMP profiler. Somehow

[5] `http://www.cecill.info/`

funny, we both are using Chrome trace viewer[6] for visualizing Gantt charts and Graphviz[7] for generating dependency graphs.

OmpSs/Extrae/Paraver and StarPU/Pajé/ViTE The Nanos runtime (OmpSs-2 programming model) implements a built-in tracer (Extrae), whose trace can be visualized in Paraver as a Gantt chart[8]. The OmpSs Mercurium compiler [126] (source to source) can compile standard OpenMP to target the Nanos runtime, so tools shall be working on standard OpenMP as well. The OmpSs-1 runtime implementation (Nanox) is capable of profiling hybrid MPI+OmpSs applications[9].

Similarly to OmpSs/Extrae/Paraver chain, StarPU [54] can generate Pajé [127] trace files to be visualized in the ViTE software [128]. StarVZ [129] is a R package and provide several visualization of task-based application executions using StarPU.

In our work, we built yet another minimal trace format optimized only for the MPC-OMP tasking runtime. We built a trace converter for visualization in Chrome Catapult viewer using an analysis pass of our framework. Additional analysis passes targetting Paraver or ViTE could be developed to benefit from their existing visualization, but was not explored as part of this thesis. In particular, the Catapult viewer for Chrome limits the visualization of traces up to 400Mo which showed to be limiting at some points. However, our tracing toolchain also comes with original analysis such as the time breakdown and communication overlap quantification as we defined in Section 4.1.2, which require virtualization of the execution post-mortem, that are not present in Paraver or ViTE.

Distributed Tools HPCToolkit [118] or TAU [119] support hybrid profiling using both OMPT and PMPI, and provides fine information on each programming model independently. In our work, we provide a joint view of hybrid OpenMP-dependent tasks and MPI communications through unified modeling, metrics, profiler, and analysis. Our approach could be integrated into existing production tools that are also capable of more advanced profiling features such as event sampling, massive data processing, or even time coherency between distributed MPI processes.

4.4 Conclusion

The lack of performances profiling tools for the task-based hybridization of MPI and OpenMP understudy lead us to develop our own toolchain as part of the MPC-OMP runtime. We introduced a unified performance modelling of task-based hybridization of MPI and OpenMP, on which we formally defined metrics: the work/idle/overhead time breakdown, the work/communication time inflation, and the overlap ratio. These metrics are well-known and fundamental for assessing on multi-threaded execution performances, but were mainly conceived for the historical bulk-synchronous parallel hybridization of MPI and OpenMP. In order to compute these metrics under the task-based hybridization understudy, we developed a new profiler into the MPC-OMP runtime. An instance of execution can be traced, and post-processed with analysis passes to compute mentioned metrics, but also to export visualizations (Gantt Charts, or the unified performance modelling graph). The profiler and its analysis passes had been widely used throughout this thesis in the results depicted in the upcoming chapters.

[6] `https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/`
[7] `https://graphviz.org/`
[8] `https://pm.bsc.es/ftp/ompss/doc/user-guide/faq-track-deps.html`
[9] `https://www.olcf.ornl.gov/wp-content/uploads/2019/08/extrae_paraver_day_2.pdf`

Chapter 5

Scheduling MPI communications as OpenMP tasks

Contents

Executing MPI communications as OpenMP tasks could improve the overlap of communications with computations [97]: The task dependency graph provides a sufficient description of the degree of parallelism so that ready-work is known to the runtime for overlapping data transfers. In addition, it could also lead to *early-bird* posting of MPI communications which "*can help alleviate a major cost of bulk-synchronous parallelism*" [71] by using tasks synchronizations over coarse threads barriers. Then, early-bird posting becomes the responsibility of the OpenMP runtime task scheduler, which may favor task paths leading to MPI communications in the TDG. However, executing MPI communications within OpenMP tasks with current production implementations (LLVM, GOMP, Open MPI, MPICH) most likely leads to poor performances [99] or even deadlocks [4]. Particularly, one issue happens when an OpenMP thread executes an MPI synchronization: the OpenMP thread ends up blocking within the MPI runtime without scheduling other OpenMP tasks: we refer to this problem as the *loss of thread issue*.

In this chapter, our contributions are two-fold. First, Section 5.1 presents extensions of the MPC-OMP runtime to tackle the loss of thread issue. Then, Section 5.2 presents improvements in the scheduler capabilities for the early-bird posting of communications, guiding decisions towards MPI communications with priorities. Most of this work had been published in the International Workshop on OpenMP (IWOMP) [130, 131].

5.1 Executing MPI Requests within MPC-OMP Tasks

In order to overlap blocking operations with useful work in a tasking environment, the idea consists of giving hand back to the task scheduler. Hence, it schedules any other independent

ready tasks until the blocking operation asynchronous completion. However, such behavior is currently not granted by programming standards and depends on the implementation. In practice, running MPI blocking operations as part of OpenMP explicit tasks most likely leads to a loss of OpenMP worker thread [4]. In Section 3.3.2, we presented the three solutions explored for blocking tasks suspension/resumption. We resume briefly each approach that ultimately motivated us to propose our design and implementation in the MPC-OMP runtime.

5.1.1 State of the art

The first solution consists of replacing blocking calls (such as `MPI_Wait`) with non-blocking test-for-completion calls (such as `MPI_Test`), and yielding to the OpenMP task scheduler (`pragma omp taskyield`) until the test passes. However, the OpenMP `taskyield` construct provides no guarantees that the scheduler will switch tasks, limiting the portability of such an approach. With existing implementations, it currently only works with the LLVM compiler, and its OpenMP runtime, and only if:

- The `taskyield` is placed on the outermost scope of the task so it is visible by the compiler when compiling the task.
- There are no dynamic stack allocations (C's variable length array, or `alloca` API)

Otherwise, in LLVM, the `taskyield` is implemented following the `stack-yield` algorithm executing a new task on top of the blocking one on the current thread stack, meaning the blocking task cannot resume until the stacked task completed, and cannot migrate between threads. In GCC, it is implemented as a "no-op" (GCC), meaning no other tasks are scheduled.

The second solution proposed by TAMPI [6] consists of automating the first solution through an additional "interoperability" layer that overrides MPI blocking operations, ensuring task suspension/resumption if executing as part of a task-based environment, such as OpenMP. The authors only provided implementation for the Nanos6 (OmpSs-2) runtime. It consists in suspending/resuming the kernel (p)thread on which the task executes using conditions and signals, and forking a new one so there is no "thread lost". Their implementation tackles the `stack-yield` stacking/migration issues, but kernel threads management introduces new overheads such as system calls; moreover, the scheduling of the blocking task is delegated from the task-based environment (OpenMP) to the operating system scheduler.

The third solution comes with the `detach(event)` clause on the `task` construct [132] introduced in 2018. It differs the completion of a task (and its dependencies release) once the associated `event` is fulfilled through the `omp_fulfill_event(event)` callback API: the task is *detachable*. This approach delegates the problem to programmers by removing blocking operations from tasks and replacing them with their non-blocking counterparts in detachable tasks. Programmers also have to raise the callback to complete the task asynchronously and ensure the correct order of execution with dependencies. It ultimately leads to the `MPI_Detach` proposal [133] (2022) that assists programmers in such a direction. However, back in 2020, at the beginning of this thesis, the `detach` clause was not well-supported in OpenMP runtimes: LLVM support was implemented in 2019 and merged in early 2020, GCC added support with release 12.1 in 2022), and MPC-OMP had no support.

5.1.2 Solutioning the loss of thread issue

While many solutions to the loss of thread issue had been proposed, they still needed to be implemented as part of the MPC-OMP environment. Inspiring ourselves from the state of the art, we designed and implemented interfaces to support the execution of MPI communications within MPC-OMP tasks.

5.1.2.1 Motivations

Stacking Tasks Using Nanos6's TAMPI implementation or the LLVM `stack-yield` algorithm, multiple tasks may be pushed onto the same thread's stack. Such an approach blocks the resumption of tasks previously stacked until the last stacked one is completed, which can raise issues. For instance, in single-producer/multi-consumer scenarios, the producer thread may end up scheduling tasks at some point. If it schedules a blocking task B on top of its original implicit task A, the task A may not resume until B is unblocked. In the best-case scenario, it only reduces the available parallelism (blocking A from producing further tasks until B is completed); but in the worst-case scenario, it can lead to a deadlock if unblocking B requires resuming A. Hence, detecting tasks that may block is essential to avoid such scenarios. However, this is not straightforward. A task may execute code from dynamic libraries that ends-up performing blocking operations at some point. Additionally, not every MPI blocking operation may retain the executing thread. For instance, this is the case when the sent data is small enough to be copied directly to the NIC or when the destination MPI process executes as part of the same POSIX process (MPC-MPI [11]) where a simple `memcpy` may be sufficient without requiring MPI processes synchronization. It is only with fine application knowledge or at execution time that the thread may decide whether it is worth suspending the current task in favor of another.

Oversubscribing Cores State-of-the-art solutions using `MPI_Detach` and Nanos6 implementation rely on kernel thread with periodic preemption. D. Tsafrir [134] showed up to 22% slowdown on a computation thread sorting an array of integers, preempted by a thread periodically repetitively making a system call to trigger the kernel scheduler. In [135], authors also show computational slowdown due to cache-related delays induced by preemption. Therefore, state-of-the-art solutions may degrade the performances of other tasks, preempting them in the middle of their execution.

5.1.2.2 Design and Implementation

To tackle the task stacking and oversubscription problems of the state-of-the-art solutions, we implemented the following design in the MPC-OMP runtime.

First, we differ in the detection of blocking task detection to programmers. Programmers have to annotate any MPC-OMP task that may be suspended with a `context(stack-size)` clause, asking the MPC-OMP runtime to attach a fixed-size stack of `stack-size` bytes to the task. Whenever such annotated tasks are scheduled, the executing thread switches to the task' stack using the `ucontext.h` library. Once the task is completed, the thread switches back to its original stack pointer and resumes execution where it stopped.

Secondly, we extended the MPC-OMP runtime with an API to suspend the current task in the middle of its execution and prohibit its scheduling until an asynchronous event (such as MPI requests completion). Full details on the API standardization and implementation have been published in [131].

Lastly, we added callbacks on the MPC-OMP runtime for progressing MPI requests and allowing the resumption of their associated OpenMP task. Right before suspending the current task, the executing thread registers the request and the task in a list. Then, on each OpenMP *schedule* event, threads progress requests using `MPI_Test` calls. If the test passes, then the task is unblocked so it may be scheduled once again. Our implementation allows progression on any thread, but only one at a time, and every request is tested once before pursuing execution. While this could have been implemented as an OpenMP Tool (OMPT), this progression callback mechanism was implemented as a built-in of the MPC-OMP runtime due to its lack of stable support for OMPT back in 2020.

The implementation is summarized in the Annex 8.1 as a one-header file TAMPI implementation for the MPC-OMP runtime, providing a portability layer across MPI runtimes for

suspending MPC-OMP tasks on `MPI_Wait` synchronizations. It had also been re-used to support the execution of blocking GPU operations in the context of OpenMP `target` tasks, with preliminary results published in [22].

5.1.2.3 Discussions

Our design and implementation come with the following benefits over existing solutions:

- A task may suspend at any time, resume on any thread, with full compatibility across codes assuming the existence of a linear stack (*i.e.* providing support to dynamic stack allocation, recursivity...).
- Suspension/resumption can be automated by simply executing blocking operations within tasks and having the MPI runtime give a hand back to the OpenMP runtime scheduler. This can be seen as an extension of the Argobots/BOLT [101] approach not only for parallel regions but also for tasking. Automation had been implemented as part of the MPC-MPI runtime, meaning that using MPC-OMP with MPC-MPI, programming can simply wrap blocking MPI operations into OpenMP tasks, and suspension/resumption occurs seamlessly. This is illustrated on the Listing 5.1 where the thread executing the `MPI_Send` line 2 may eventually yield to the task line 4 if blocking on an MPI synchronization.
- The MPI request progression on each OMPT *schedule* event ensures up-to-date knowledge of ready tasks before the OpenMP scheduler makes a new scheduling decision. In addition, it likely reduces interferences over state-of-the-art solutions with no preemption in the middle of their execution.

```
1 # pragma omp task context(stack-size) untied
2    MPI_Send(...); // blocking operation
3
4 # pragma omp task
5 {
6    // independent work
7 }
```

Listing 5.1: MPC-OMP and MPC-MPI program with automated task suspension/resumption

However, our design and implementation also come with at least the following drawbacks:

- Programmers have to guess and annotate tasks that may block with a new `context` clause, so our runtime executes them using a dedicated stack (otherwise, they are executed on top of the thread original stack). Automatically managing OpenMP task stacks without the need of a new programmer annotation and evaluating induced performance overheads is yet to be evaluated more precisely.
- The use of a dedicated task' stack may have unnecessary overheads: (1) annotated tasks may never block (for instance, MPI eager protocol), meaning that stacking them onto the parent thread stack would have been acceptable, and (2) stack-less execution using LLVM `detach` implementation removes stack management costs (allocations, context switches).
- The `context` clause comes with a `stack-size` parameter raising the problem of optimizing it. Too small, execution may stack overflow; too big, we may consume unnecessary memory. In [136], authors propose an automated verification of stack size requirements for C programs at compile-time. Their approach is limited by Variable-Length Array (VLA) or `alloca`, which can hardly be predicted at compile-time. Outside these cases, their prediction is convincing and could lead to the automation of optimal `stack-size` parameters by the compiler.

- Our implementation relies on the `<ucontext.h>` library that is deprecated. It was introduced in POSIX.1-2001 as a portable interface for threads execution context at the user-space level. The POSIX.1-2008 made this interface deprecated, recommending applications to be rewritten using POSIX threads[1]. Yet, practice showed the library to be portable across the recent Fugaku (2020) and CEA-HF (2022) supercomputers.

5.1.3 Summary

In this section, we have motivated and presented our design and implementation to tackle the loss of thread issue. It enables the execution of MPI requests as part of OpenMP tasks. In such a way, the overlap of MPI synchronization with independent OpenMP tasks may occur seamlessly, as the task dependency graph already provides the parallelism available to the task scheduler. This work enables the task-based hybridization of MPC-OMP with any MPI implementation, which leads us to explore the early-bird posting of communications by favoring tasks leading to MPI operations in the following section.

5.2 Early-bird Posting with Tasks Scheduling

In the previous section, we have introduced extensions towards automating the overlapping and progression of MPI communications through the OpenMP task scheduler. However, a second motivation of the task-based hybridization of MPI and OpenMP is the *early-bird* posting of MPI communication by scheduling OpenMP tasks leading to them in the TDGs. Let us motivate furthermore the benefits of early-bird communication posting with the following minimal example.

5.2.1 Motivations

The reference benchmark in the HPC tasking community is the matrix Cholesky decomposition. Given a matrix A, the benchmark computes the lower triangular matrix L so that $A = L.L^T$; which can also be used to resolve the $A.x = b$ system. Implementations of the literature mostly rely on executing dense algebra kernels (from BLAS, LAPACK, CuBlas...) into tasks. It makes Cholesky a great example of tasking for such calculation that are sometimes on the core of scientific simulation. In 2018, J. Schuchart et al. [99] implemented a tiled Cholesky factorization mixing MPI and OpenMP using tasks. We use it as a motivational example for early-bird communication posting using OpenMP task scheduling.

Fig. 5.1 depicts a subgraph of the Cholesky factorization TDG, and its scheduling on 2 MPI processes of 2 OpenMP threads each from a real execution. Tasks are scheduled following MPC-OMP default "First-In, First-Out" (FIFO) scheduling policy: first-ready tasks are scheduled first. In this minimal example, such policy led to 61% idleness on cores. If task work and communication time are assumed constant, a portion of the idleness could be reduced by earlier posting of the MPI send requests, as shown on the alternative execution presented Fig. 5.2 (which is not a real execution). In this alternative scheduling, we favored the execution of tasks performing MPI send requests and their predecessors. Once yellow tasks (L) and (H) are completed, three tasks (P), (M), and (J) become ready. In the first-case scenario following a FIFO policy, (J) is favored over the communication task (P) on process 1. In parallel, the process 0 thread ends up idling: there are no more ready tasks, as local pending tasks depend on remote data sent by (P). Delaying the communication (P) on process 0 lead process 1 to idle. In the second case scenario, (P) is favored over (J): the send request is posted as soon as it is ready. Hence, process 0 spend less time idling for data to arrive and can schedule tasks (F) and (C) earlier. In the end, idleness is reduced to 36% thanks to the earlier send request posting.

[1] `https://man.netbsd.org/NetBSD-8.0/i386/ucontext.2`

It must be noted that favoring only tasks performing MPI communications is not sufficient: favoring their predecessors as well led to early posting of the task (O) for instance.

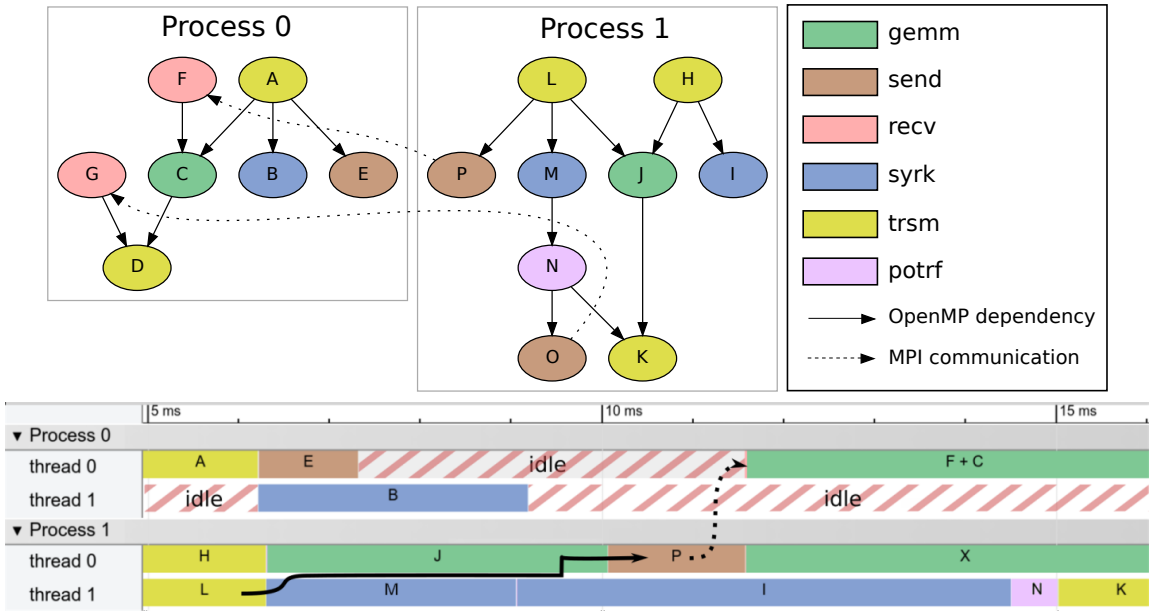


Figure 5.1: Top: sub-graph of a distributed blocked Cholesky factorization mapped onto 2 MPI Ranks (matrix size: 2048x2048 with tile of size 512). Bottom: Gantt chart with 2 threads per MPI process (real execution).

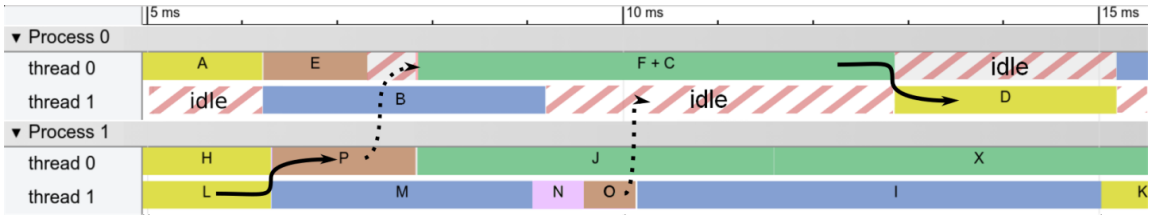


Figure 5.2: Alternative scheduling for the graph in Fig. 5.1 (not real execution)

Contributions In this section, we present how we enabled early-bird posting of MPI communication through the MPC-OMP tasking runtime scheduler. Section 5.2.2 presents a current limitations of existing OpenMP runtimes that limits the scheduler visions of the TDG, and therefore, early-bird communication scheduling. Section 5.2.3 presents MPC-OMP runtime extensions so tasks can be assigned priorities interpreted by the runtime scheduler, to favor tasks leading to communications. Finally Section 5.2.4 presents tasks priority heuristics with different level of automation for application programmers to guide the runtime scheduler towards the early-bird posting of MPI communications.

5.2.2 The Issue of Task Throttling

Task Throttling is a runtime mechanism originally motivated to reduce tasking operational and memory overheads [93]. Once a threshold is reached, producer threads stop producing and start consuming tasks instead. This mechanism limits the vision of the runtime scheduler of the future of the execution [116], making it impossible to perform early-bird communication task scheduling as tasks may not even be known by the runtime scheduler.

In [93], authors proposed to prune task creation and sequentialize their execution at run-time, using dynamic information such as the task depth in the control flow graph (CFG). Both GCC and LLVM runtimes implements a threshold bounding the number of ready-tasks that can co-exist which was developed in the context of OpenMP independent task model (version 3.0), as an efficient solution to bound the memory consumption. It is set to $64 \times nthreads$ in GCC and 256 in LLVM. Though, it can be totally disabled in LLVM[2] deferring the memory bounding responsibility to the application programmer.

In case of dependent tasks, ready-tasks throttling threshold not only limits the vision of the runtime scheduler; it is also insufficient to bound memory consumption overhead: as many successors tasks may be created but not marked as ready. Therefore, we extended the MPC-OMP runtime with a threshold on the total number of tasks that can co-exist (ready or not). This threshold can be configured through the `MPCFRAMEWORK_OMP_TASK_MAXIMUM` environment variable set to $10,000,000$ by default. With a task descriptor of $512bytes$ and $16bytes$ per edge, this default values ensure at most $5Go$ of memory for task descriptors. This represents 4% of the available DRAM on AMD EPYC processors used in Exa or Titan supercomputers. A patch has also been submitted for adoption into LLVM[3].

5.2.3 Scheduling Tasks with Priorities

In order to control task scheduling, the OpenMP specifications provide a `priority(p)` clause on the `task` construct, taking an integer parameter h. This clause is a *hint*, and the standard recommends implementations to favor the scheduling of tasks with higher priority values. It also mentions that programs should not rely on this mechanism to ensure the correct order of execution as these are only recommendations, and implementers remain free to do as they please.

As we have shown in our minimal example on the Cholesky decomposition, not only tasks performing MPI communications should be favored, but also their predecessors in the task dependency graph. On this point, the standard provides no means or guidelines for propagating over the task dependency graph: under current specifications (and its production implementations GCC/LLVM), it is the programmer's responsibility to set priorities for every task of its application. Note that back-propagating priorities by the runtime to predecessors may increase furthermore task creation overheads.

5.2.3.1 A Design for Dependent Tasks Priority

In order to assist application programmers on favoring tasks leading to MPI communications, and limit induced overheads, we extended the interpretation of `priority` hint with three parameters: (`VALUE`, `PROPAGATION`, `SYNCHRONY`). Each parameter is defined with environment variables for the entire execution and they cannot be safely changed during the execution.

- `VALUE` ∈ `{ZERO, INF, COPY}` - controls the task priority. `ZERO` means the hint is ignored and the priority is set to 0, `INF` means tasks with non-zero hint will have an `INT_MAX` priority, and `COPY` means tasks priority will be set to the hint.
- `PROPAGATION` ∈ `{NONE, EQUAL, DECREMENT}` - controls how the priority should be propagated to predecessors. `NONE` means the priority will not be back-propagated, `EQUALS` means the priority will be copied to predecessors recursively, `DECREMENT` means the priority will be decremented and copied to predecessors recursively.
- `SYNCHRONY` ∈ `{SYNCHRONOUS, ASYNCHRONOUS}` - controls when the priority should be back-propagated (if `PROPAGATION` is not `NONE`). `SYNCHRONOUS` means it should be propagated

[2] `https://reviews.llvm.org/D63196`
[3] `https://reviews.llvm.org/D158416`

on task creation by the producer thread that is between lines 16 and 17 of Listing 3.1. `ASYNCHRONOUS` means it should be propagated opportunistically any time by the runtime.

Algorithm 1 summarizes this design and provides the priority $P(T)$ for a given task T of hint $H(T)$ depending on the parameters `VALUE` and `PROPAGATION`. The routine *Successors* returns the set of successors for the given task.

Remark Configuration (`COPY`, `NONE`, `_`) corresponds to GCC and LLVM interpretation of the `priority` clause.

Algorithm 1 Task Priority

Input: Task T
Output: Integer P(T) the priority of T

```
1:  function P(T)
2:      if PROPAGATION = NONE or Successors(T) = ∅ then
3:          if VALUE = ZERO then
4:              return 0
5:          else if VALUE = COPY then
6:              return H(T)
7:          else                                          ▷ If VALUE = INF
8:              return INT_MAX
9:      else if PROPAGATION = EQUAL then
10:         return max({P(S) | S ∈ Successors(T)})
11:     else                                    ▷ If PROPAGATION = DECREMENT
12:         return max({P(S) − 1 | S ∈ Successors(T)})
```

5.2.3.2 Implementation of Priority Support in the MPC-OMP Runtime

Back in 2019, the original MPC-OMP runtime used to ignore the `priority` hint at all. This lead us to propose an implementation following the design we just introduced.

First, to sort-out tasks following their priorities, we implemented a red-black tree data structure following Chapter 13 of [137]. It is a binary search and self-balancing tree that can perform operations (search, insert, delete) with $O(log_2\ n)$ time and $O(n)$ memory complexity, where each node of the tree is a *list* of tasks with the same priority. Fig. 5.3 depicts tree example where the root node contains the tasks T_3 and T_8 with priority 5. Whenever a task must be elected from a red-back tree, the runtime first selects the list with most priority (list 9 in the example), and then pops a task following a "First In, First Out" (FIFO) or "Last In, Last Out" (LIFO) strategy, that can be configured at launch with an environment variable.

We replaced MPC-OMP original task lists with red-black trees of 4 types that can be placed on different topological levels as shown in Fig. 5.4.

- `NEW` trees placement is configurable: in the example, they are placed per NUMA domain. It stores ready tasks with no predecessors.
- `SUCCESSORS` trees are placed on the lowest topological level (one per core), storing ready tasks successors of previously completed tasks on that core to favor temporal locality as in [138],
- `TIED` trees are also placed on the lowest topological level, storing suspended tied tasks,
- `UNTIED` trees placement is configurable: in the example, they are placed per socket. It stores suspended untied tasks.

Algorithm 2 summarizes which tree is elected whenever a task has to be queued.

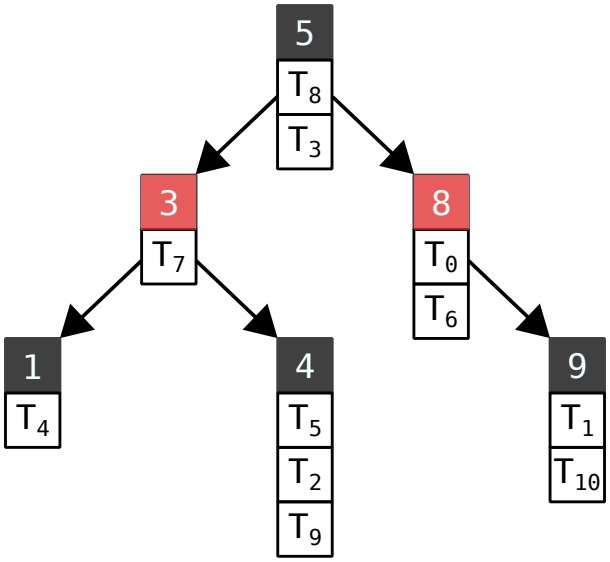


Figure 5.3: Red/Black Tree with nodes as task list of the same priority

Algorithm 2 Task Queuing into the Topological Red Black Trees

Input: Task T

```
1:  if T has started then
2:      if T is tied then
3:          Queue in TIED tree of the current core
4:      else
5:          Queue in UNTIED tree of the current core
6:  else
7:      if T has no predecessors then
8:          Queue in NEW tree of the current core
9:      else
10:         Queue in SUCCESSORS tree of the core on which T last predecessor completed
```

Note that each couple (C, `TYPE`) with C a core and `TYPE` a red-black tree type identifies a unique red-black tree in the topology in the path from C to the root node: for instance on Fig. 5.4, (3, `NEW`) identifies the `NEW` tree of NUMA 2. Algorithm 3 presents the decision taken by a thread executing on core C whenever a new task is scheduled. The *pop* routine pops the task from the list with most priority of the red-black tree identified by the couple (C, `TYPE`). The *steal* routine steals work into trees of type `TYPE` from core C.

Algorithm 3 Task Schedule

Input: Core C

```
1: return pop (C, SUCCESSORS) if not null
2: return pop (C, NEW) if not null
3: return steal(C, NEW) if not null
4: return steal(C, SUCCESSORS) if not null
5: return pop (C, TIED) if not null
6: return pop (C, UNTIED) if not null
7: return steal(C, UNTIED)
```

Regarding the `ASYNCHRONOUS` propagation mode, we implemented it so that any thread can

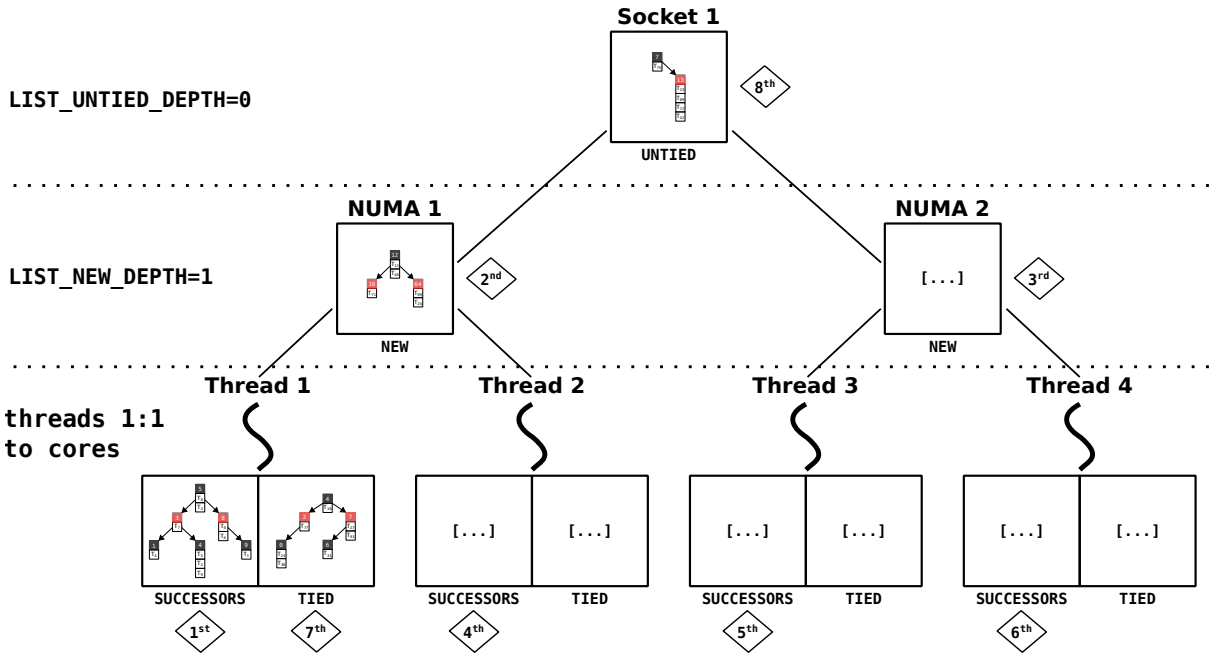


Figure 5.4: Hierarchical Red/Black Tree Scheduling

propagate priorities in the TDG during idle periods, but restricting to only one thread at a time. The propagation algorithm is depicted on Algorithm 4 and is executed by the first thread idling. It consists of two-step alternating: (1) finding leaves of the TDG from a root task and (2) computing and back-propagating priorities to parent tasks in the TDG. On line 20, we do not allow ready tasks to change priority. The reason is the *work-first* principle: the propagation only occurs when threads are idling. Updating a task priority would mean popping it off queues and reinserting it back, which would prevent another worker from scheduling the task during the priority update. Following the same principle, our implementation of this algorithm also embeds a stopping criterion: if enough tasks are ready, the propagating thread breaks.

5.2.3.3 Evaluation

In order to evaluate impacts on performance of our task priority management, we made the micro-benchmark in Annex 8.3. On this benchmark, each thread creates $1,000,000$ empty tasks with a priority set using the system pseudo-random number generator. Note that this benchmark is among the worst case scenario, as (1) we are running empty tasks, hence emphasizing runtime structures contentions; and (2) setting a random priority $p \in [0; 2^{31} - 1]$, so its very likely that every tasks has its own priority, hence a red-black tree with 16,000,000 nodes.

We compile it with GCC 13.2.0 and executed it on 16 AMD EPYC 7H12 cores with 16 threads bound 1:1 to cores. We set the `OMP_MAX_TASK_PRIORITY` environement variable to `INT_MAX` ($2^31 - 1$) and execute in two modes to compare against GCC 13.2.0 and LLVM 16.0.0 runtime implementations:

- `no-priority` with (`VALUE`, `PROPAGATION`, `SYNCHRONY`) = (`ZERO`, `NONE`, `SYNCHRONOUS`). This way, every tasks are assigned a 0 priority and scheduled using the default LIFO policy.
- `with-priority` with (`VALUE`, `PROPAGATION`, `SYNCHRONY`) = (`COPY`, `NONE`, `SYNCHRONOUS`). This way, task copy the passed priority value and are inserted into the thread red-black tree.

We performed 5 executions and Table 5.1 reports minimum/median/maximum execution times. A first observation comparing runtimes, is that MPC-OMP tasking management seems to have

Algorithm 4 Priority propagation (single-thread, during idle periods)

```
1:  Variables
2:      List D, U                                   ▷ D, U stands for DOWN, UP
3:      Task T, S, P
4:
5:  procedure PROPAGATE(ROOTS)          ▷ Propagate priority from existing root tasks
6:      D = [], U = []                                                ▷ Empty lists
7:      for T in ROOTS do
8:          Push T to D tail
9:      while D is not empty do                          ▷ Step (1) Search for leaves
10:         T = Pop D head
11:         if T has successors then
12:             for S in Successors(T) do
13:                 if S is not VISITED then
14:                     Mark S as VISITED
15:                     Push S to D tail
16:         else
17:             Push T to U tail                    ▷ Store the leaf T to the list U
18:         while U is not empty do          ▷ Step (2) Climb-up the TDG from leaves
19:             T = Pop U head
20:             if T is not queued then   ▷ T is already queued, not updating its priority
21:                 for P in Predecessors(T) do
22:                     Update P priority                   ▷ Based on T priority
23:                     Push P to U head
```

much more overheads than LLVM and GCC respectively with a 3.18x and 1.78x slowdown in the `no-priority` mode. A second observation comparing the two modes:

- LLVM is whether deadlocking or very slow when enabling priorities, as we couldn't complete the execution.
- GCC has no overheads when enabling priorities. Our interpretation comes from its task throttling limiting only 64 tasks per threads to co-exist, which likely reduces priority management costs.
- MPC-OMP has a 2.6x slowdown when enabling priorities, as tasks requires sorting in red-black trees.

	Time (in s.)			Time (in s.)			Time (in s.)		
	LLVM 16.0.0			GCC 13.2.0			MPC 5f91c6		
	Min.	Med.	Max.	Min.	Med.	Max.	Min.	Med.	Max.
no-priority	6.09	7.10	7.11	3.68	3.99	4.81	12.60	12.70	13.00
with-priority	N/A	N/A	N/A	3.70	3.93	5.19	33.05	33.57	34.38

Table 5.1: Performance of the micro-benchmark Annex 8.3

5.2.4 Communication-Aware Task Scheduling Strategies

As the Cholesky motivational example illustrated, favoring tasks performing MPI communications and their predecessors could improve performances. Using the new MPC-OMP priority-based

scheduling infrastructure, we evaluated four priority heuristics with various intrusivity in user-code and runtimes in that direction. This work has been mostly published in [130] with a few more recent evaluations in the end of the section.

5.2.4.1 Guiding the Task Scheduler with Priorities

Manual 1 (MA-1) The runtime is configured with (`COPY`, `NONE`, `_`), and tasks performing `MPI_Send` communications are annotated with `priority(1)` on the user-code. It means ready tasks performing `MPI_Send` communications will always be chosen first against any other ready tasks.

Manual 2 (MA-2) The runtime is configured with (`COPY`, `EQUAL`, `SYNCHRONOUS`), and tasks performing `MPI_Send` communications are annotated with `priority(1)` on the user-code. It means ready tasks on a path from any roots to an `MPI_Send` communications will always be chosen first against other tasks.

Semi-Automatic (SA) The runtime is configured with (`INF`, `DECREMENT`, `SYNCHRONOUS`), and tasks performing `MPI_Send` communications are annotated with `priority(1)` on the user-code. It means that given a set of ready tasks, the closest task to any `MPI_Send` communications will be favoured over other tasks.

Fully-Automatic (FA) The runtime is configured with (`ZERO`, `DECREMENT`, `ASYNCHRONOUS`), and tasks performing `MPI_Send` communications are not annotated the `priority` clause. Instead, the MPI runtime notifies the OpenMP runtime whenever it performs an `MPI_Send` operation. Then, the MPC-OMP runtime saves a task *profile* and automatically sets the priority to `INT_MAX` for future tasks with the same profile. The profile is made of tasks information such as its function pointer, its data size, its properties (tiedness, final-clause, undeferability, mergeability, if-clause), its parent task identifier, its number of predecessors, and its number of successors. As opposed to **SA**, **FA** does not require the programmer to annotate any tasks, but on the other hand, it has two limitations:

- The first tasks performing `MPI_send` communication (and their predecessors) will not be favored, as our OpenMP runtime has no clue about a task's content until it executes it once.
- The profile matching may lead to false-negative; for instance, the number of successors may not be fully known if tasks are being consumed as soon as they are produced.

Illustration Fig. 5.5 illustrates the task ordering on a Cholesky matrix decomposition, with the default scheduling heuristics (FIFO) with the third heuristic (SA): the redder the task, the earlier it is scheduled. Diamond tasks represent tasks performing `MPI_send` communication. As it can be seen on the SA ordering, the scheduler favors tasks leading to these communications, while the FIFO heuristic lead to a breadth-first scheduling. These figures illustrates that the SA heuristics manages to favor not only tasks performing `MPI_Send` but also their predecessors.

5.2.4.2 Evaluation

Fig. 5.6 shows the impact of cores spreading between MPI and OpenMP on the performance, varying scheduling heuristics, using the median over 20 executions. Note that the amount of computational tasks on each configuration remains constant: there is precisely 2,829,056 computational tasks distributed across MPI processes, while the number of communication increases with the number of MPI ranks. Table 5.2 shows the total number of point-to-point MPI requests in total (receive and send), with one request per OpenMP task.

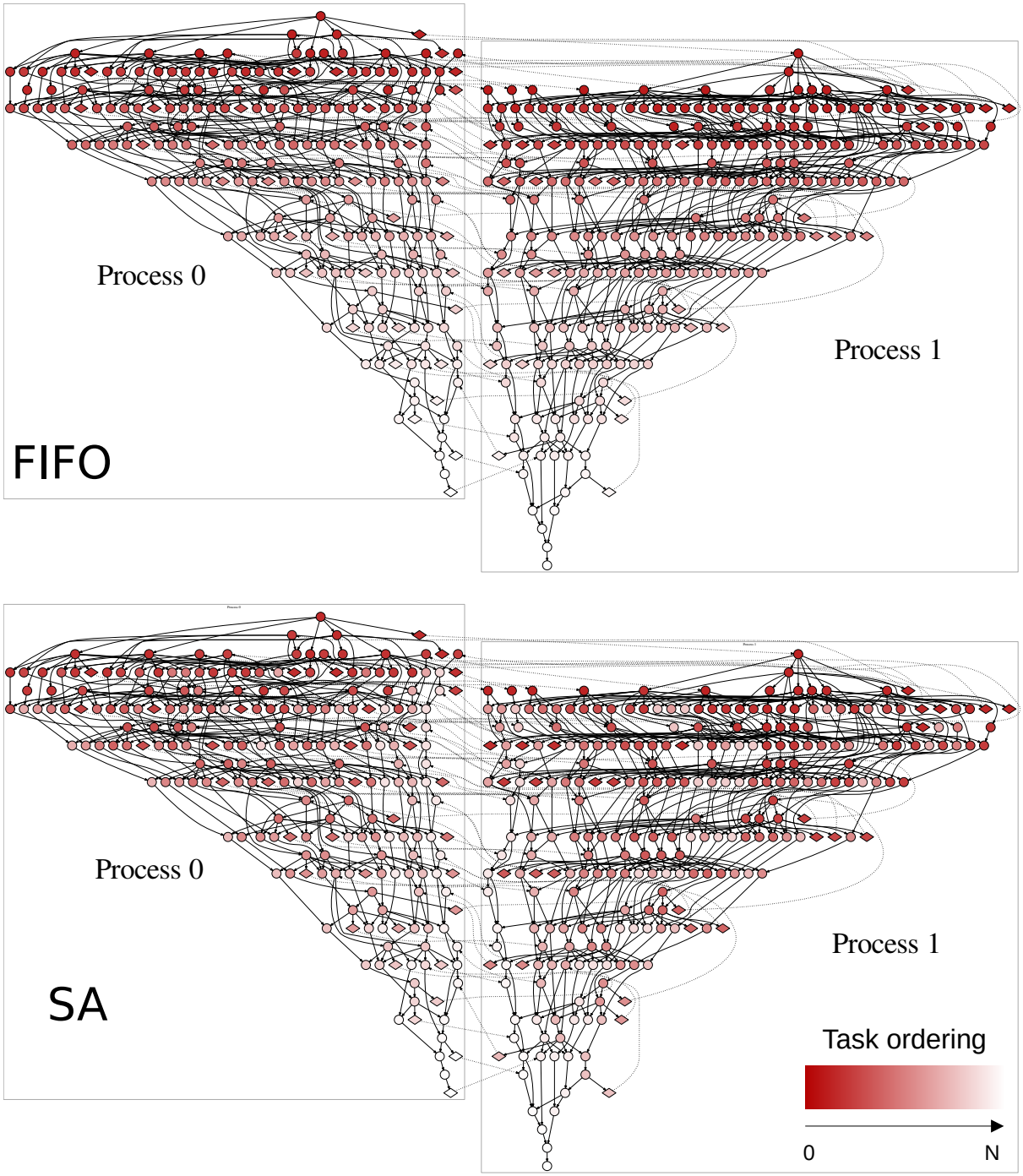


Figure 5.5: Task Ordering (the redder, the earlier schedule) depending on the heuristic used, on two MPI processes for $n = 6,144$ and $m = 512$. The SA heuristic favors `MPI_Send` (diamond tasks) and their predecessors, while FIFO lead to breadth-first scheduling

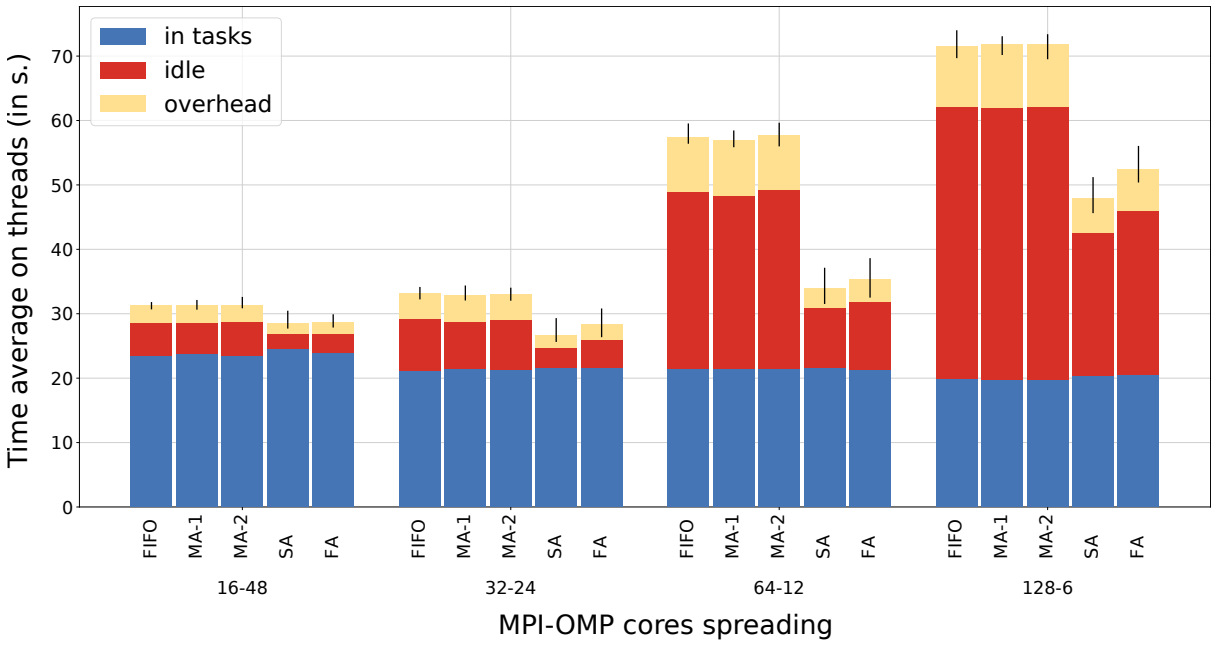


Figure 5.6: Cholesky factorisation time varying heuristics, and the MPI/OMP cores spreading on 16 Skylakes nodes (e.g., 16-48 stands for 16 MPI ranks of 48 threads each) - with a matrix of size $n = 131,072$, and blocks of size $m = 512$ - Using MPC-MPI and MPC-OMP

Table 5.2: Number of point-to-point communication tasks in Fig. 5.6 runs

Cores spreading (MPI-OpenMP)	16-48	32-24	64-12	128-6
P2P communication tasks (overall)	388.624	640.632	892.640	1.372.880

These results show that the scheduling heuristic used can have a significant impact on performance in the presence of MPI communications.

Overall Performances The MA-1 and MA-2 heuristics do not improve performances over the reference FIFO policy, but SA and FA do, respectively by 20% and 14%. It means that only favoring ready communication tasks is not sufficient, but their predecessors must be favoured as well. Note that the performance gain is directly reflected in idleness reduction, as MPI processes spend less time synchronizing due to earlier communication posting. Even though FA improves performances, it is never as good as SA for the two limitations mentioned previously.

Idleness As a general observation, increasing the number of MPI processes seems to increase the average idleness, likely due to the increase in the number of MPI synchronizations as heuristics SA and FA attenuate this phenomenon.

Work time On each spreading, threads are bound 1:1 compactly to the NUMA domain; which means on the 16-48 spreading, the 48 OpenMP threads are bound on 2 different 24-cores NUMA domain. Inter NUMA domain memory accesses likely is the source of the work time inflation observed [9], as work stealing may lead tasks' successors to migrate between NUMA domains. On the other spreadings, the work time remains about the same, independently of the heuristic used.

Overhead While overheads seem slightly higher on the FIFO, MA-1 and MA-2 heuristics, the actual reasons are not fully understood yet. A potential explanation could be the contention while accessing runtime internal data structures: the more idleness, the more concurrency on

accesses. This behavior is specific to the MPC-OMP runtime, where threads almost never sleep at the kernel level and perform a lot of active waiting.

5.2.4.3 Another More Recent Evaluation

The previous experiment were conducted at the beginning of this thesis in 2021. We reproduced the experiment in 2023, but using an updated version of MPC-OpenMP, and switching from MPC-MPI to OpenMPI 4.1.5. We only reproduced the configuration with 32 MPI processes of 24 OpenMP threads each. Table 5.3 depicts the results: the first column is the scheduling parameters, the second and third column are the TDG execution (makespan) and discovery time; and the last three columns provide the time breakdown averaged on threads. We provide the median time for each value and its standard deviation over the 20 runs.

Mode	Makespan	Discovery	Work	Idle	Overhead
no priority (+ FIFO)	28.45 ± 1.63	0.44 ± 0.06	20.83 ± 0.76	7.96 ± 1.85	0.18 ± 0.01
no priority (+ LIFO)	28.05 ± 1.58	0.45 ± 0.05	20.70 ± 0.82	7.67 ± 1.96	0.18 ± 0.01
SA (+ LIFO)	27.04 ± 1.72	0.61 ± 0.05	21.72 ± 1.60	5.71 ± 3.03	0.18 ± 0.03
FA (+ LIFO)	27.25 ± 1.71	0.51 ± 0.05	20.74 ± 1.57	6.62 ± 2.35	0.38 ± 0.20

Table 5.3: Cholesky Factorization Performances varying scheduling parameters with MPC-OMP 2023 and OpenMPI 4.1.5. Times are given in seconds (s.)

These results validate previous results obtained in 2021, but gains on idleness and overall performances are smaller. A first reason is that we switched from MPC-MPI to OpenMPI, that ultimately lead us to detect a performance issue in the MPC-MPI runtime related to communications route creation. A second reason is that overheads also significantly reduced since 2021, likely due to the MPI runtime switch but also with MPC-OMP runtime improvements over the 2 years.

The SA configuration inflates the discovery time (about 36%) over the `no priority` configurations, explained by the synchronous priority propagation in the TDG by the producer thread. It also inflates the work time by about 5% which reasons remains to be investigated. The FA configuration slightly inflates the discovery time (about 14%) over the `no priority` configurations, likely because of runtime lock contention between the asynchronous priority propagation occurring in parallel of the discovery.

In both the SA and FA versions, performances gain (makespan) are explained by reducing idleness on cores, most likely because favoring tasks performing `MPI_Send` operations and their predecessors reduces idleness from remote data missing, as shown in the motivating example previously.

5.3 Related Works

OpenMP as a Low-Level Parallel Runtime OpenMP is used as a low-level back-end runtime for intra-node parallelization of higher-level programming models such as Kokkos [50] or Raja [49], PGAS (XcalableMP [139]), or Domain Specific Languages/Abstraction (DSL/DSA) (Devito [140], Nablab [43]). They provide a higher abstraction that allows programmers to write codes without low-level parallel considerations. In the specific case of DSL using MLIR [141] compilers, OpenMP is easily targetable, and merging/suspending tasks can be achieved. Our extensions on task context and suspensipn could be used in the code generated as a portable and low-overhead solution for suspending/resuming tasks until the completion of an external asynchronous event.

Scheduling Tasks in a Non-Uniform Memory Space Non-Uniform Memory Accesses (NUMA) architectures are now widely adopted by GPPs equipping supercomputers. This had lead to several works on the scheduling of tasks on such architectures [138, 142, 143]. Olivier et al. [142] showed that hierarchical task scheduling with respect to the hardware topology can significantly impact performances. Debres et al. [143] proposed a joint scheduling and memory allocation algorithm, mixing a memory allocator with a work-stealing strategy, for task-parallel programs execution on NUMA systems. Virouleau et al. [138] presented runtime strategies for scheduling dependent OpenMP tasks on NUMA architectures. In particular, they draw the conclusion that both the topology and the initial task data placement impact performances.

User-Level Threads Scheduling While GNU/Linux and its (kernel) pthread implementation is now equipping most supercomputers, other implementations were proposed. In particular, the SunOS [144]. (later known as "Solaris") provided a support of the `PTHREAD_SCOPE_PROCESS` pthread scope; said differently, it implemented a user-level threading pthread-compliant library with lightweight overheads.

The Parallel Multi-threaded Machine (PM2[4]) is a distributed multi-threaded programming environment. Its original design (1995) consisted in Marcel (its threading library) and Madeleine (its communication library) [145]. Similarly to MPI, a PM2 program is replicated across a cluster of compute node, where each PM2 process can communicates to one another. Back in 1997, the PM2 environment already proposed the co-scheduling of user-level threads to reduce threading management overheads.

The Multi-Processor Computing runtime (MPC) [11] implements virtual processors (VPs) with kernel threads, which can schedule MPI processes, OpenMP threads or even pthreads as user-level threads (MPC threads) on top of it. MPC threads are slightly more rigid than OpenMP tasks, and re-using MPC threads for the tasks execution context management would have implied important modifications of MPC threading layer: for instance, load balancing of MPC threads on VPs is much more limited than OpenMP tasks work-stealing strategy between OpenMP threads [146].

Argobots [85] is a low-level threading and tasking library. Authors conducted an important set of evaluations on user-space execution context management costs, and proposed several optimizations (such as « Not saving the context of terminating ULT ») that we could be integrated into MPC to reduce execution context management costs. The Bolt [101] OpenMP runtime implements its threads using Argobots. The task execution context extension we proposed could be implemented with Argobots fibers, but bridging the two worlds (OpenMP tasks, and Argobots fiber) would likely raise a few difficulties.

5.4 Conclusion

By nesting MPI communications into OpenMP tasks, programmers can differ the responsibility of overlapping, progression, and early-posting of MPI communication to the OpenMP task scheduler. However, current standard and implementations have limitations that makes it hard to achieve in practice. In this chapter, we have presented several extensions (in the OpenMP standard and runtimes) towards the automation of interactions between MPI and OpenMP to improve task-based hybridization.

In shorts:

- Tasks can be attached a user-level threads so they may suspend at any point of their execution without blocking the executing threads as per-before.
- A standard interface had been proposed for suspending/resuming tasks until the completion of an asynchronous operation. As opposed to existing interfaces, it goes towards standard

[4] `https://pm2.gitlabpages.inria.fr/`

OpenMP interface to be used by runtimes, hiding the suspension/resumption burden from applications developers.

- MPI progression callbacks can be registered into the runtime and raised at different execution point (such as scheduling points) of the runtime scheduler. This approach ensures weak progression of MPI communications in-between the task scheduler decisions, and the resumption of suspended tasks.
- We proposed a design and an implementation for task priorities in OpenMP, that not only favor annotated tasks (as currently) but also their predecessors. Performances results on the heuristics evaluations have shown that priority propagation to predecessors is necessary to ensure actual early-bird posting of MPI communications.

Ultimately, all these extensions lead us to show slight performance improvements on a Cholesky factorization, as we compared scheduling heuristics with various level of automation for programmers. Results show that early-bird posting slightly improve performances over naive breadth-first or depth-first scheduling, by reducing idleness that was mostly likely due to remote data being sent lately.

Chapter 6

Enabling Dependent Tasking in HPC Applications

Contents

While state-of-the-art solutions enable working task-based hybridization, few applications migrated towards a task-based composition. At least one attempt led to convincing performances with the porting of the Cholesky factorization by J. Schuchart and al. [99]. Other attempts whether failed to implement functional applications due to interoperability issues [4], had to tinker application codes [116] to sequentialize communications or added coarse barriers losing potential communication overlap [147]. The lack of applications comes from the *difficulties* to reach functional and high-performance codes using a task-based composition of the current standards and their runtime implementation.

In the introduction of this manuscript, we declined these difficulties in three sources all tied together: *Performances*, *Profiling*, and *Programming*. Our work investigates these difficulties jointly and provides pieces of answers to each. Chapter 4 presented an hybrid profiler providing answers *profiling* and *performances* difficulties. Chapter 5 proposed an important set of extensions to improve both *programming* and *performances* aspects, that we evaluated on small benchmarks. In the Chapter 6, we pursue this work on applications more realistic of scientific simulation codes. Firstly, we present the porting of the HPCG benchmark and the LULESH proxy-application; with a few standard extension proposal during our journey that shall ease future applications porting. Then, preliminary evaluations of the ported applications guided us towards investigating impacts on the task dependency graph *discovery* on performances; and yet again to extend the standard and runtime.

6.1 Porting Irregular Applications with Task-based Programming

Production scientific codes are usually complex and closed-source codes. Therefore, computer scientist bases their research on *benchmarks* and *proxy-applications*. Benchmarks are usually small codes to assess raw computational power. Proxy-applications model production scientific simulations extracting their critical components to smaller and simpler open-source codes. Proxy applications aim to explicit scientific simulation needs to help design hardware, programming models, and end-codes.

The High-Performance Conjugate Gradient (HPCG [148]) benchmark and the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH [149]) proxy-application are two mainstream HPC use-cases Both original codes were parallelized with OpenMP workshare loops (`# pragma omp parallel for`) and MPI non-blocking and non-persistent communications. This section introduces in detail each application, and presents our porting experience towards the task-based composition of MPI and OpenMP, restricting our use of the standard as follows:

- The entire program runs into a `parallel` and a `single` region.
- The `single` thread discovers the application parallelism using the `task` and `taskloop` construct and the `depend` clause. Therefore, the task graph discovery occurs on a single *producer* thread concurrently and in parallel with its execution by every thread (including the producer). Note that using the `depend` clause on the `taskloop` construct is not allowed by the standard specifications, but our use follows the proposal made by M. Maroñas et al. [150].
- The computation is distributed through MPI, and communications are nested into `task` regions, inserted into the task dependency graph as any other tasks.

Remark At some point, we considered extending our use of the OpenMP standard to the `teams` construct, typically placing a single MPI process and OpenMP runtime per compute-node and one team per NUMA domain. This way, we could 1/ dampen the number of MPI processes furthermore and 2/ remove shared-memory messages. Yet, it implied important programming efforts; while 1/ the number of MPI processes has never shown to be limiting in the experiments of this thesis, and 2/ MPI implementations are already capable of automatically detecting and optimizing shared-memory message passing [151]. Therefore, we keep extra levels of partitioning as future optimization work, which could be, for instance, through the OpenMP `team` construct or new constructs such as hierarchical tasking [152, 153].

6.1.1 Conjugate Gradient (HPCCG)

The High-Performance Linpack[1] (HPL) [154] and the High-Performance Conjugate Gradient[2] (HPCG) [148] are two complementary benchmarks for assessing on a supercomputer performance. Both benchmarks resolve linear system $A.x = b$ and provides a *Flops/s* metric corresponding to the number of *floating point* operations (additions, multiplications) performed per second. However, HPL performs *dense* algebra (few zeroes in matrices) algebra with regular memory accesses; while HPCG performs *sparse* algebra (elements are mostly zeroes) with irregular memory accesses. This makes the HPL *compute-bound* and HPCG *memory-bound* on most systems; respectively meaning respectively that HPL execution time is mostly determined by the CPU speed, while HPCG execution time is mostly determined by memory accesses speed. In term of raw performances, it results in HPL computing 85x (on Frontier) and 27x (on Fugaku) more Flops/s on HPL against HPCCG.

[1] `https://netlib.org/benchmark/hpl/`
[2] `https://hpcg-benchmark.org/`

The HPCG benchmark implements the Conjugate Gradient method depicted on Algorithm 5. It consists of a sequence of dot products (dot), scaled vector addition (axpby), and sparse matrix-vector multiplication (SpMV). Each operation is implemented with element-wise loops and originally parallelized using the `parallel for` construct as shown on Listing 6.1.

```
1 # pragma omp parallel for
2 for (Index_t i = 0; i < n; i++)
3     local_result += x[i] * y[i];
```

Listing 6.1: HPCCG' dot product implementation

Algorithm 5 Conjugate Gradient Algorithm

Input: Matrix A, Vector b, Vector x_0
Output: Vector x - solution of $A.x = b$

function CG(A, b)
  $r_0 \leftarrow b - A.x_0$
  if r_0 is small-enough **then** ▷ for a given norm
    return x_0
  $p_0 \leftarrow x_0$
  $k \leftarrow 0$
  loop
    $\alpha_k \leftarrow \frac{r_k \cdot r_k}{p_k^T . A . p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k . p_k$
    $r_{k+1} \leftarrow r_k + \alpha_k . A . p_k$
    if r_{k+1} is small-enough **then**
      return x_{k+1}
    $\beta_k \leftarrow \frac{r_{k+1}^T \cdot r_{k+1}^T}{r_k^T \cdot r_k^T}$
    $p_{k+1} \leftarrow r_{k+1} + \beta_k . p_k$
    $k \leftarrow k + 1$

6.1.1.1 Task-based Porting of HPCCG

The task granularity is a critical parameter to reach high performances with task-based applications: finely tuning the granularity is necessary to balance tasking management costs with expressed parallelism [91, 155]. To do so, we cut each of the three algebraic operations into sub-operations over vector/matrix blocks (or "tiles"), and assigning one or more blocks to tasks. We cut operations using two parameters $(T_1, T_2) \in \mathbb{N}$ while preserving the correct order of execution by inferring dependencies on block data accesses, as shown on the following equations. Let

$$\begin{aligned}(A_{i,j}) \in M_n(\mathbb{R}), (x_i) \in \mathbb{R}^n, (y_i) \in \mathbb{R}^n, (w_i) \in \mathbb{R}^n,\\ T_1 \in [\![1 \,..\, n]\!] \text{ and } B_1 = ceil(n/T_1),\\ T_2 \in [\![1 \,..\, B_1]\!] \text{ and } B_2 = ceil(B_1/T2)\\ (\alpha, \beta) \in \mathbb{R}^2,\end{aligned}$$

then dot products are computed as

$$\sum_{i=1}^{n} x_i y_i = \sum_{t_1=0}^{T_1-1} \underbrace{\sum_{i=t_1 B_1+1}^{min(n,(t_1+1)B_1)} x_i y_i = \alpha}_{\texttt{\# pragma omp task depend(in: x[}t_1 B_1\texttt{], y[}t_1 B_1\texttt{]) depend(out:}\alpha\texttt{)}} \tag{6.1}$$

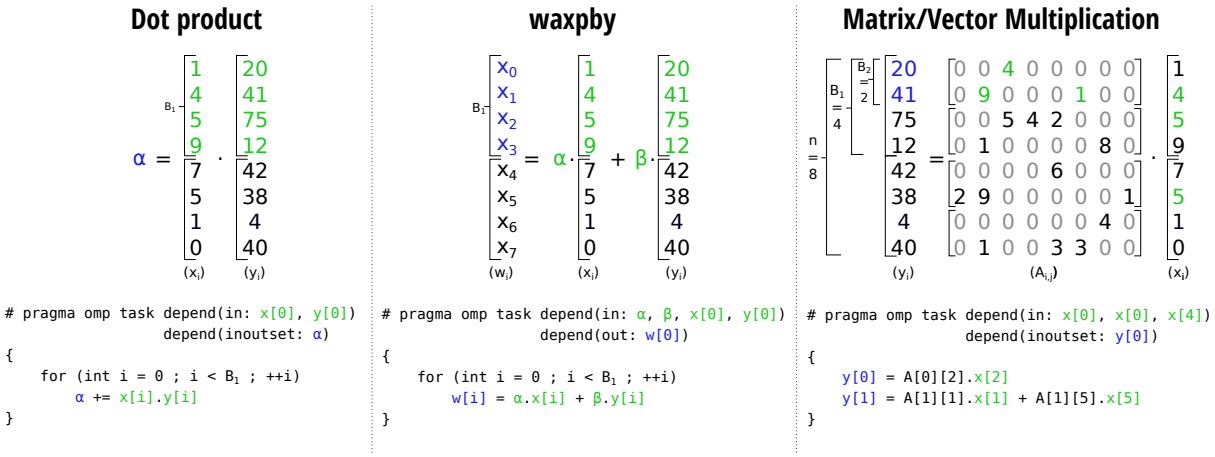


Figure 6.1: HPCCG tasks for $n = 8$ and $T_1 = T_2 = 2$

waxpby as

$$\forall t_1 \in [\![0 \,..\, T_1 - 1]\!], I_1 := t_1 B_1 + 1 \text{ and } I_2 := min(n, (t_1+1)B_1)$$
$$\underbrace{\forall i \in [\![I_1 \,..\, I_2]\!], w_i = \alpha x_i + \beta y_i}_{\texttt{\# pragma omp task depend(in: x[}t_1B_1\texttt{], y[}t_1B_1\texttt{]) depend(out: w[}t_1B_1\texttt{])}} \tag{6.2}$$

SpMV with

$$\forall i \in [\![1 \,..\, n]\!], z_e(A,i) = \{j \in [\![1 \,..\, n]\!] \mid A_{i,j} \neq 0\}$$
$$z_b(A,i) = \{t_1 B_1 \mid t_1 B_1 \leq j < (t_1+1)B_1 \mid j \in z_e(A,i)\}$$

as

$$\forall t_1 \in [\![0 \,..\, T_1 - 1]\!], \forall t_2 \in [\![0 \,..\, T_2 - 1]\!],$$
$$I_1 := t_1 B_1 + t_2 B_2 + 1$$
$$I_2 := min(n, min(t_1 B_1 + (t_2+1)B_2, (t_1+1)B_1))$$
$$\underbrace{\forall i \in [\![I_1 \,..\, I_2]\!], y_i = \sum_{j \in z_e(A,i)} A_{i,j} x_j}_{\texttt{\# pragma omp task depend(in: x[k] so that } k \in \underbrace{\bigcup_{i \in [\![I_1 \,..\, I_2]\!]} z_b(A,i)}_{\texttt{not real code}}\texttt{) depend(inoutset: y[}t_1B_1\texttt{])}} \tag{6.3}$$

Remark Our implementation does not balance the number of non-zeroes element between tasks. Though, each line of the matrix has consecutive 27 non-zero elements per line in the reference version of HPCCG, meaning that

$$\forall i \in [\![1..n]\!], \leq Card(z_b(A,i)) \leq 27$$

Sets $z_b(A,i) \mid [\![1..n]\!]$ are indices of the vector blocks for which the matrix has non-zero elements in an SpMV multiplication: they are computed only once on the initialization.

Fig. 6.1 illustrates each task for $n = 8$ and $T_1 = T_2 = 2$, and depicts the first task of each operation, that is, for $t_1 = t_2 = 0$. In this example, computing the first block of y, $z_b(A,0) \cup z_b(A,1) = \{0,4\}$ are the two x blocks for which A has columns with non-zero elements, hence the dependencies on $x[0]$ and $x[4]$.

Fig. 6.2 depicts a task dependency graph on a single iteration for the reference `parallel-for` implementation on 2 threads, and our task-based implementation for $(T_1, T_2) = (2, 1)$. The

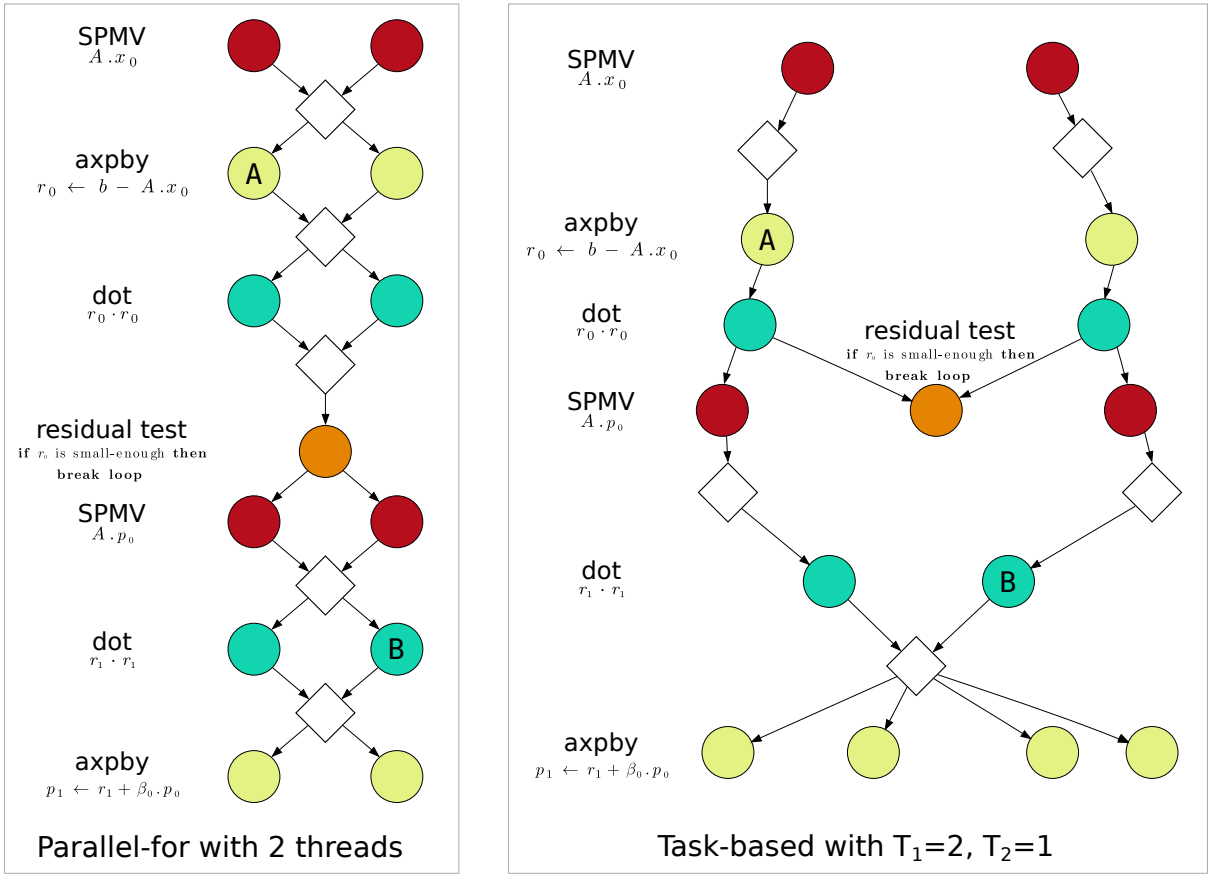


Figure 6.2: Conjugate Gradient Graph for the `parallel-for` and `task` version, on a 1 iteration

diamond white nodes are control-flow synchronization tasks, modeling, for instance, a `for` loop barrier or an `inoutset` reduction. Using tasks over work-share loops, we express finer synchronizations through dependencies over barriers: the task-based version allows the concurrent execution of `A` and `B` while the work-share version does not. In the context of MPI, such approach can lead to earlier posting of communication removing coarse synchronization of work-share constructs.

Task type	FLOPs	Median grain (ms.)	Performance (GFLOPs/s)
dot	$2.\frac{n}{T_1}$	12.3	0.43
waxpby	$3.\frac{n}{T_1}$	25.9	0.30
SpMV	$2 \times 27.\frac{n}{T_1.T_2}$	517.3	0.27

Table 6.1: HPCCG: the number of FLOPs performed per task, and the median grain measured on 16 AMD EPYC 7H12 cores for $n = 41,943,040$, $T_1 = 16$ and $T_2 = 1$

6.1.1.2 Expressing Irregular Tiled Dependencies

The main programming difficulty faced when porting the HPCCG application to task-based OpenMP was on expressing irregular dependencies. While expressing regular dependencies is rather straight-forward with OpenMP (dot 6.1 and waxpby 6.2), expressing tiled-irregular dependencies is more challenging.

```
1  std::list<int> indices;
2  int k2 = MIN(k+B2, (k/B1+1)*B1, n);
3  for (int i = k; i < k2; ++i)
4      for (int j = row[i]; j < row[i+1]; ++j)
5          indices.insert(inds.end(), col_idx[j]/B1*B1);
6
7  auto it = indices.begin();
8  # pragma omp task depend(iterator(i=0:indices.size()), in:  x[*it++]) depend(out:  y[k/B1*B1])
9  {
10     for (int i = k; i < k2; ++i) {
11         y[i] = 0;
12         for (int j = row[i]; j < row[i+1]; ++j)
13             y[i] += data[j] * x[col_idx[j]];
14     }
15 }
```

Listing 6.2: Matrix Vector Multiplication redundant and tiled dependencies

```
1  std::set<int> indices;
2  [...]
```

Listing 6.3: Matrix Vector Multiplication tiled dependencies, filtering redundant dependency

Listing 6.2 is our first implementation of the SpMV irregular dependencies using the Compressed Sparse Row (CSR) matrix format. With the current 5.2 specifications, indices from the union $\bigcup z_b(A,i)$ on SpMV (6.3) must be computed and placed into a list, iterating on it using OpenMP `iterator` within a `depend` clause, as shown on Line 7 for Listing 6.2. Lines 1 to 4 are executed once on the matrix initialization, placing column block indices with non-zero elements consecutively into a C++ `std::list`, for the matrix lines k to $k2$. On the first SpMV block ($k = 0$) of the example Fig. 6.1, the list would be $[0, 0, 4]$ as it depends on the x vector blocks starting at indices 0 and 4. The block 0 appearing twice in the list is said *redundant*. Redundant dependencies are not programming errors of the OpenMP standard, and the runtime is capable of filtering them out. However, it unnecessarily and repetitively stresses out the runtime on every iteration. On a second implementation, we simply replaced the C++ list with a `std::set` as shown on Listing 6.3; filtering-out redundancies once for all on matrix initialization in the

user-code, so there are no redundancies passed to the runtime on a task construct.

6.1.1.3 The Impact of Redundant Dependencies

Task dependencies management is a significant source of overheads for processing dependent tasks with OpenMP [155]. We experimented to evaluate its impact on performance. Table 6.2 presents results on 16 AMD EPYC 7H12 cores on a matrix of size $n = 262,144$ with $T_1 = 16$ and $T_2 = 1$.

Columns present the number of dependencies passed to the runtime, the task graph creation time, and the task graph execution.

Row (a) presents measurements for the reference `parallel-for` HPCCG implementation (hence with no explicit tasks nor dependencies). Row (b) uses task with redundant and *per-scalar* dependencies, meaning one distinct dependency is expressed for each memory address accessed by the task. Row (c) uses task with redundant and *tiled* dependencies, meaning there is one dependency expressed for each memory address accessed by the task, pointing to the vector/matrix block of that address (and so, likely many redundancies). It corresponds to the previous Listing 6.2. Row (d) uses tasks with no redundant and tiled dependencies, meaning one dependency is expressed for each vector/matrix block accessed by the task. It corresponds to the previous Listing 6.3.

	Dependencies passed	Discovery	Execution
(a) parallel-for	N/A	N/A	0.21s.
(b) tasks, redundant per-scalar dependencies	605,705,049	158s.	158s.
(c) tasks, redundant tiled dependencies	438,992,697	33s.	33s.
(d) tasks, non-redundant tiled dependencies	19,641	0.02s.	0.23s.

Table 6.2: HPCG Performances on Dependencies Expression

Using per-scalar task dependencies leads to a 752x slowdown over the reference version due to higher task discovery costs. Tiling dependencies slightly improved performances reducing the slowdown to 157x ; but filtering redundancies was the key to reaching reasonable performances (1.09x slowdown). The runtime has several less order of magnitude dependencies to process, resulting in faster task creation.

6.1.1.4 Automating Filtering with an API Extension

As previous results suggest, removing redundancies and tiling irregular dependencies is crucial for performances. OpenMP provides the `depobj` interfaces for pre-allocating dependencies, that can then be passed several time to the same task through the `depend` clause. While it looks like a good candidate for such dependency scheme (for instance, filtering redundancies and tiling only once for all at matrix initialization), its current restrictions currently makes it impossible in practice. Therefore, we propose the following extensions and restriction relieves so that Listing 6.4 becomes standard-compliant.

- (1) We remove the following constraint on the `depobj` construct: "*A depend clause on a depobj construct can only specify one locator*". On the listing, writing `x[col_idx[j]/`B`_1*`B`_1]` (line 3) and `y[k]` (line 4) on the same `depobj` construct is illegal currently.
- (2) We extend the `depobj` construct so it can have multiple dependency types. On the listing, writing on the same `depobj` construct both `in:` (line 3) and `out:` (line 4) is illegal currently.
- (3) We remove the 'unique' property from the `iterator` modifier on a `depend` clause. On the listing, writing two `iterator` in the same `depend` clause (line 3) is illegal currently.

Implementation and evaluation of these extensions are kept as future work, and here are a few suggestions for implementers:

- Compiler: (1) redundant dependencies should be filtered-out at compile-time if possible; and (2) the compiler shall inform the runtime with the total number of dependencies n, and an upper-bound on the number of redundancies $m \in [0, n]$.
- Runtime: filtering-out redundancies consists in building a set from a sequence of n integer with m redundancies at most. A high-quality implementation should adapt to special cases such as $m = 0$ and $m = n$ (yet very common in regular applications). As a matter of fact, results presented Table 6.2, SpMV tasks had $70,116,861$ redundancies for 2 or 3 unique dependencies. Refining tasks with $T_1 = 512$ and $T_2 = 16$ (optimal grain), we had $68,758$ redundancies for 1 or 2 unique dependencies.

```
1 omp_depend_t obj;
2 # pragma omp depobj(obj)
3         depend(iterator(i=k:k+B2), iterator(j=row[i]:row[i+1]), in:  x[col_idx[j]/B1*B1])
4         depend(out:  y[k])
5 [...]
6 # pragma omp task depend(obj)
7 [...]
```

Listing 6.4: Sparse Matrix / Vector Multiplication Task with no restrictions on depobj (deps. per block filtering redundancies)

6.1.1.5 Studying the Impact of Task Granularity

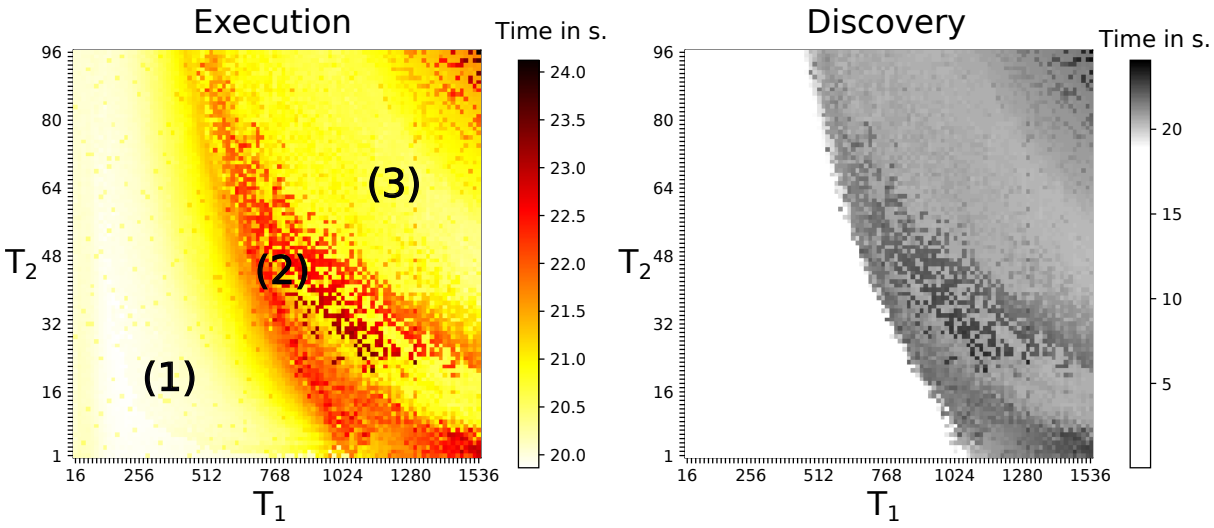


Figure 6.3: HPCCG Task Graph Execution (left) and Discovery (right) time on 16 AMD EPYC 7H12 cores

Tasks granularity can have a significant impact over performances. Too coarse, it may lead to a lack of parallelism and important idleness; too fine, management overheads may deteriorate performances [7]. Fig. 6.3 presents a task grain study on HPCCG, it shows the task graph execution (left) and its discovery (right) time. The discovery is the time from the first to the last task creation, occuring on a single producer thread concurrently of its execution by any threads (including the producer). The execution corresponds to wall-clock time from the first task schedule to the completion of the last task. Each point of the figure correspond to a single run on 16 AMD EPYC 7H12 cores. We set $n = 41,943,040$ occupying 62% of the NUMA domain

memory capacity over 32 iterations, varying grain parameters $T_1 \in [16; 1536]$ with a step of 16 and $T_2 \in [1, 96]$ with a step of 1. It means there are few coarse tasks on the bottom-left points and many fine tasks on the top-right points. Colors indicate the time: the lighter, the shorter. We observe that both the graph execution and discovery time significantly vary with the number of tasks. More precisely, we observe 3 areas:

- (1) is the *execution-bound* area; tasks discovery is slower than their parallel execution, and the total time is bound by tasks execution.
- (2) is the *execution/discovery frontier* area; tasks discovery and their parallel execution time are close, and the total time may be slowed down due to tasks being discovered too slowly.
- (3) is the *discovery-bound* area; tasks discovery time and their parallel execution time are close, and the total time is mostly bound by task discovery, and threads end up idling waiting for work to be discovered.

For this HPCCG problem, the best performances are reached in area (1), which gives a range of efficient task grains for this problem size. We observe a *rebound effect* between areas (2) and (3). Going through the frontier area (2), the tasks are getting consumed as soon as they are created. It makes the producer thread prune dependency edges since predecessors no longer exist on successors' discovery. Even though more tasks are discovered in the end, this accelerates the task graph discovery as there are fewer edges created. After the reddish region ($(T_1; T_2) = (1, 200; 64)$), there are almost no edges generated: a sign that we reached the *discovery-bound* area (3). Discovery is slightly faster than within the frontier area (2); until there are too many tasks (upper-right-most corner at $(T_1; T_2) = (1, 536; 96)$), and refining tasks furthermore would only slow down the execution time.

6.1.2 Simplified Hydrodynamic Simulation (LULESH)

While benchmarks primarily focus on performances, they are not representative of 1M+ lines production codes needs in term of programming. To respond to this limitation, the Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) [156] laboratory from the United States Department of Energy (DOE) provided an important set of proxy applications, parallelized hybridizing MPI and OpenMP, which are more realistic of production application codes. The original purpose was co-designing machines, codes, and programming models towards exascale, that ultimately lead to the Frontier machine. As part of this thesis, we study the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH [149]) proxy application. This proxy application is modeling the core of ALE3D [157], a 3D multi-physics simulation software capable of simulating multiple physical phenomena such as heat transfers or chemistry models.

6.1.2.1 A brief introduction on LULESH

LULESH models an *hydrodynamic simulation* of materials motion subject to forces over an *unstructured mesh*. The simulation duration and the mesh can be configured with command line arguments. We briefly introduce these concepts that guided our porting to the task-based model.

Unstructured Mesh An unstructured mesh is a set of *nodes* connected into *elements*. Each node is a multi-dimensional object embedding some properties such as a position vector (x, y, z), a velocity (v_x, v_y, v_z) or a mass m. The mesh represents a discrete three-dimensional spatial domain, with nodes being points and hexahedrons elements representing volumetric units. On LULESH, the mesh data structure is built-in for simplicity purposes: each element is made of 8

```
1  typedef struct {       /* a node data structure */
2      double x, y, z;    /* position */
3      double vx, vy, vz; /* velocity */
4      double m;          /* mass */
5  } node_t;
6
7  typedef struct {       /* a mesh data structure */
8      node_t nodes[n];   /* set of nodes */
9      int elements[m][8]; /* set of elements pointing to 8 nodes */
10 } mesh_t;
```

Listing 6.5: Simplified LULESH mesh data structure

nodes represented as an indirection array, as depicted on Listing 6.5. Internally, the LULESH benchmark uses a uniform cartesian grid, but its still represented as an unstructured mesh so researches on it are applicable to more complex irregular meshes used in production simulations. The mesh is a cube partitioned into sub-cubes distributed on MPI processes; therefore the application can only runs using a cubical number of MPI processes. On each MPI rank, local mesh nodes and elements lists are ordered along axes so that nodes and elements at position (x, y, z) in the cube are placed at index $x.s^2 + y.s. + z$ in lists.

An Hydrodynamic Simulation Zooming into the code, it looks something similar to the code Listing 6.6. Line 3, the simulation iterates until a certain amount of iterations `max_iter` had been executed. Then each iteration starts with a global reduction (`MPI_Allreduce`) to compute a dynamic time step `dt` on line 5 to 7. Afterward from line 9 to 19, there is multiple mesh-wide computational loops iterating, so the simulation approximates hydrodynamics equations solutions discretely onto elements. In the original version provided by LLNL, each computational loop is parallelized using `# pragma omp parallel for` construct, and 9 out of the 38 loops also are annotated with the `nowait` clause. Then at some point of the iteration (line 22 to 30), *frontier* nodes are exchanged between MPI processes. Each MPI process can have up to 27 neighbors connected through faces, edges or nodes; and only one layer of ghost nodes is used. Finally after exchanging, there is a few more computational loops (line 32) before the next iteration starts.

Parameters The command line argument `-i` controls the number `max_iter` of simulation iterations, and `-s` the mesh size. For a given size s distributed on p MPI processes, the total number of elements is $p \times s^3$ and the total number of nodes is $p \times (s+1)^3$ with each MPI process working on s^3 elements and $(s+1)^3$ nodes.

6.1.2.2 Reference parallel-for version

The original LLNL version of the code relies on a *fork-join* programming model; the computation is parallelized through OpenMP `parallel for` (line 10 and 15 of Listing 6.6) and distributed with MPI communications outside of OpenMP constructs: the two programming models are used separately in the code. It also comes with two reports [149, 158] presenting how the code must be used to remain representative of production user needs. For instance, constraints include the mesh data structure representation, the loop structure, and some extra computation. In [159], authors show that LULESH original code is poorly optimized but reports constraints limit the applicability of their optimizations. Though, their *global allocation of temporary work arrays* optimization is compatible. It consists of preallocating memory buffers instead of repetitively calling `malloc` and `free` within the inner-most loops. Hence, we backported it, which increased performances by about 24% on our configuration. In future evaluation, this optimized version is used as the `parallel-for` reference.

```
mesh_t mesh;
double dt;
for (int iter = 0 ; iter < max_iter ; ++iter)
{
    /* compute the time step (MPI_Allreduce) */
    double local_dt = compute_local_timestep();
    MPI_Allreduce(&dt, &local_dt, sizeof(double), MPI_MIN, MPI_COMM_WORLD);

    /* nodes-wise loop: update nodes properties */
    # pragma omp parallel for
    for (int i = 0 ; i < mesh.nodes.size() ; ++i)
        work_on_nodes(mesh.nodes[i], dt);

    /* elements-wise loop: resolve partial differential equations */
    # pragma omp parallel for
    for (int i = 0 ; i < mesh.elements.size() ; ++i)
        work_on_elements(mesh.elements[i], dt);

    /* more loops [...] */

    /* Exchange frontier nodes */
    MPI_Irecv(recv_buffer, ...);

    Pack(send_buffer, mesh);
    MPI_Isend(send_buffer, ...);

    /* more send/recv [...] */

    MPI_Waitall(...);
    Unpack(rbuffer, mesh);

    /* more computational loops [...] (38 loops overall) */
}
```

Listing 6.6: Simplified code of LULESH simulation

```
1  mesh_t mesh;
2  double dt;
3  for (int iter = 0 ; iter < max_iter ; ++iter)
4  {
5      # pragma omp task depend(out:  dt)
6      {
7          double local_dt = compute_local_timestep();
8          MPI_Allreduce(&dt, &local_dt, sizeof(double), MPI_MIN, MPI_COMM_WORLD);
9      }
10
11     # pragma omp taskloop depend(in:  dt) depend(out:  mesh.nodes[i]) firstprivate(iter)
           num_tasks(tnl)
12     for (int i = 0 ; i < mesh.nodes.size() ; ++i)
13         work_on_nodes(mesh.nodes[i], dt);
14
15     # pragma omp taskloop depend(in:  dt, mesh.nodes[...])  depend(out:  mesh.nodes[...])
           firstprivate(iter) num_tasks(tel)
16     for (int i = 0 ; i < mesh.elements.size() ; ++i)
17         work_on_elements(mesh.elements[i], dt);
18
19     /* more loops [...] */
20
21     # pragma omp task depend(out:  mesh.nodes[...])
22     {
23         MPI_Recv(rbuffer, ...);
24         Unpack(rbuffer, mesh);
25     }
26
27     /* more send/recv [...] */
28
29     # pragma omp task depend(in:  mesh.nodes[...])
30     {
31         Pack(sbuffer, mesh);
32         MPI_Send(sbuffer, ...);
33     }
34
35     /* more computational loops [...] (38 loops overall) */
36 }
```

Listing 6.7: Simplified task-based code of LULESH simulation

6.1.2.3 Task-based Porting

With a particular attention on the two reports constraints, we ported LULESH to the task-based model composition understudy. The porting follows lessons learned from the HPCCG porting: dependencies are tiled, and redundancies are filtered once and for all on the mesh initialization.

Computational Loops We transformed the reference version `parallel for` computational loops (lines 10 and 15 of Listing 6.6) to dependent-tasks generating loops (lines 11 and 15 on Listing 6.7) with consecutive nodes or elements in the original loop computed in the same task. The number of tasks generated from node-wise and element-wise loops can be configured with a command line parameter, respectively `-tnl` and `-tel`, using a `num_tasks` clause on the `taskloop` construct. Dependencies are inferred from the loop breakdown into tasks with respect to the sequential data accesses (read/write) on the mesh, ensuring a correct order of execution while still exposing as much parallelism as possible. The annex Table 8.1 presents the average grain of each task (in µs.) measured on 24 Intel(R) Xeon(R) Platinum 8168 cores for $s = 384$, $tel = tnl = 1024$, and $i = 16$ compiled with GCC 12.2.0.

Communications We split frontier communications into multiple MPI persistent requests with balanced sizes. Similarly to loop parallelization, consecutive nodes in lists are communicated in the same MPI request. Each MPI request is placed into an OpenMP task with appropriate dependency to ensure its correct order of execution. The number of requests per frontier can be configured with the command line argument `-rpf` and `-rpe` respectively for the number of requests per face and per edge (nodes-frontier are always made of 1 request). Note that both parameters were fixed to 1 in the rest of this thesis to fall back to the reference application coarse-grain communication pattern (`-rpf=1` and `-rpe=1`). Fine study on communication grain is kept as future work.

Minimal Example Fig. 6.4 illustrates the mesh partitioning into tasks on a minimal example. It shows a 2-dimensional slice of the domain along the x and y axes on 8 MPI ranks (only 4 shown). In this example, parameters value are `-s 2`, `-tnl 2`, `-tel 2`, `-rpf 1` and `-rpe 1`. Listing 6.7 illustrates a minimal porting to dependent tasking of Listing 6.6. On line 14, the `mesh.nodes[...]` refers to irregular dependencies (to be understood as: "the elements' nodes on which the task is computing"). The `num_tasks` clause specifies the number of tasks on loops, and the `grainsize` is inferred from it similarly to the `for schedule(dynamic) num_tasks()` construct.

6.1.2.4 Studying the Impact of Task Granularity

As HPCG, we conducted an evaluation on the impact of task granularity on overall performances. Fig. 6.5 presents results as the execution time and the task dependency graph discovery time varying grain parameters on 16 AMD EPYC 7H12 cores. We set $s = 256$ occupying 72% of the NUMA domain memory capacity over 16 iterations, varying grain parameters $tel = tnl$ varying in $[16, 2048]$ with a step of 16, increasing and refining tasks from left to right. We make three following observations on this result:

- The same three areas as of HPCCG appears, annotated with (1), (2) and (3) on the figure x-axis.
- There are some specific grains where the graph discovery appears to be very slow, for instance, on $tel = 1,168$ or $tel = 1,520$.
- On coarse grain, the execution time is inflated to $60s$ and reduces to about $30s$ when refining tasks grain.

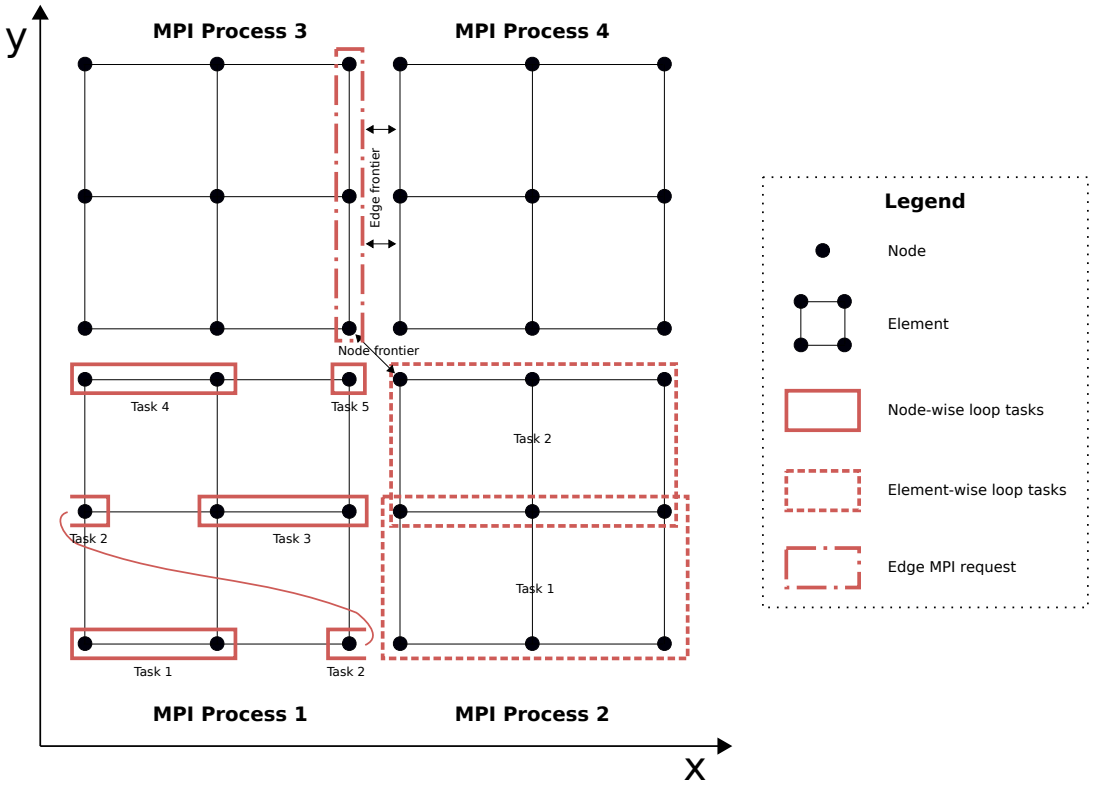


Figure 6.4: LULESH Slice View along the (x, y) axes

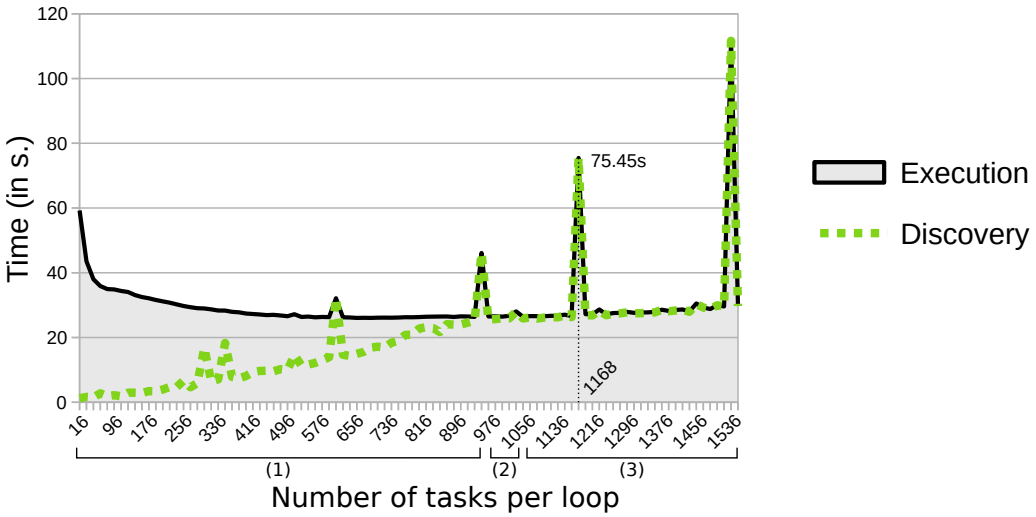


Figure 6.5: Execution and Discovery time of the task dependency graph of LULESH, with -s 256 -i 16 varying tel=tnl parameters, using `xor_6_2` hashing, on 16 AMD EPYC 7763 Cores

The first observation finds the same reasons as HPCCG: the runtime automatically prunes edges from already consumed predecessors. The second observation is explained in the following paragraph. The third observation is studied in Section 6.2.

6.1.2.5 The Impact of Dependency Hashing on the Graph Discovery

The MPC-OMP runtime relies on an hash table to build tasks precedence constraints. The runtime uses the uthash library [160] that implements hash table methods summarized in [161]. The table hashes variables *virtual memory addresses* passed through the `depend` clause with tasks and infer precedence constraints following the standard specifications on dependency modifiers (`in`, `out`, `inout`, `mutexinoutset`, `inoutset`). Listing 6.8 is the data structure mapped by a dependency variable. The same dependency management relying on address hashing is used in GOMP, KMP, and Nanos6 (OmpSs-2); but GOMP/KMP implements their own hash tables, and Nanos6 uses the C++ `std::unordered_map`. We instrumented the MPC-OMP runtime and found out that get/put operations on this hash table were much slower on the low-performance peaks of the figure. Hence, we reproduced the same experiment varying the hashing function on Fig. 6.6. We compare the original MPC-OMP hash function also used in KMP (`xor-6-2`), GOMP (`xor-0-32`), Nanos6/MPC-OMP (`div-8`), and a Jenkins (`lookup2`) hash functions whose definition are provided in Listing 6.9.

Left-side figures report the parallel execution time (gray filled area), the single-thread discovery time (green dotted line), and the portion of the discovery time corresponding to hash map get/put accesses (blue dashed/dotted line). Right-side figures report the average number of collisions in the hash table for a single LULESH iteration.

The hash table was configured to resize automatically if more than 10 collisions for a given key occur, to preserve $O(1)$ access costs. Nevertheless, the `uthash` implementation will not resize if it detects that the hash function is not a good fit for the key domain losing $O(1)$ access time[3]. For some task grains, such as $1,158$ on `xor-6-2`, the number of collisions goes above 500 meaning that the function is not a good fit and that the runtime stopped resizing the hashmap; hence the peak number of collisions. This ends up slowing down hash map operations and therefore the graph discovery, which is ultimately bounding the execution time.

Another important result of these experiments is that in MPC-OMP, even for a "good fit" hash function (`lookup2`), the hash map accesses represent from 6% to 52% of the task graph discovery costs (37.17% on average), making it a critical component of the tasking runtime for dependency management overheads. Note that these costs is related to the number of expressed dependencies: reducing it while preserving the correct order of execution would reduce hash map accesses overheads.

```
1  /* KMP */
2  uintptr_t xor_6_2(void * addr)
3  {
4      uintptr_t v = (uintptr_t) addr;
5      return (v >> 6) ^ (v >> 2);
6  }
7
8  /* GOMP */
9  uintptr_t xor_0_32(void * addr) {
10     uintptr_t v = (uintptr_t) addr;
11     v ^= v >> (sizeof (uintptr_t) / 2 * CHAR_BIT);
12     return v;
13 }
14
15 /* Nanos6 and original MPC-OMP */
16 uintptr_t div_8(void * addr) {
17     return ((uintptr_t)addr) >> 3;
```

[3] https://github.com/cea-hpc/mpc/blob/master/src/MPC_Common/include/uthash.h#L1120-L1125

```
struct {
    /* address for the dependency */
    void * addr;

    /* mutex for accessing attributes */
    mutex_t lock;

    /* last 'out' or 'inout' task for this address */
    task_t * out;

    /* list of 'in' and 'inoutset' tasks for this address */
    task_list_t * ins, inoutset;

    /* last task that depend on this address (used for redundancy check) */
    int task_uid;

    /* the uthash handle */
    UT_hash_handle hh;
};
```

Listing 6.8: Hash table entry for dependency management

```
}

/* Jenkins Lookup 2 (uthash implementation) */
uintptr_t xor_0_32(void * addr) {
    uintptr_t hashv;
    HASH_JEN(&addr, sizeof(void *), hashv);
    return hashv;
}
```

Listing 6.9: Dependency Hashing Functions

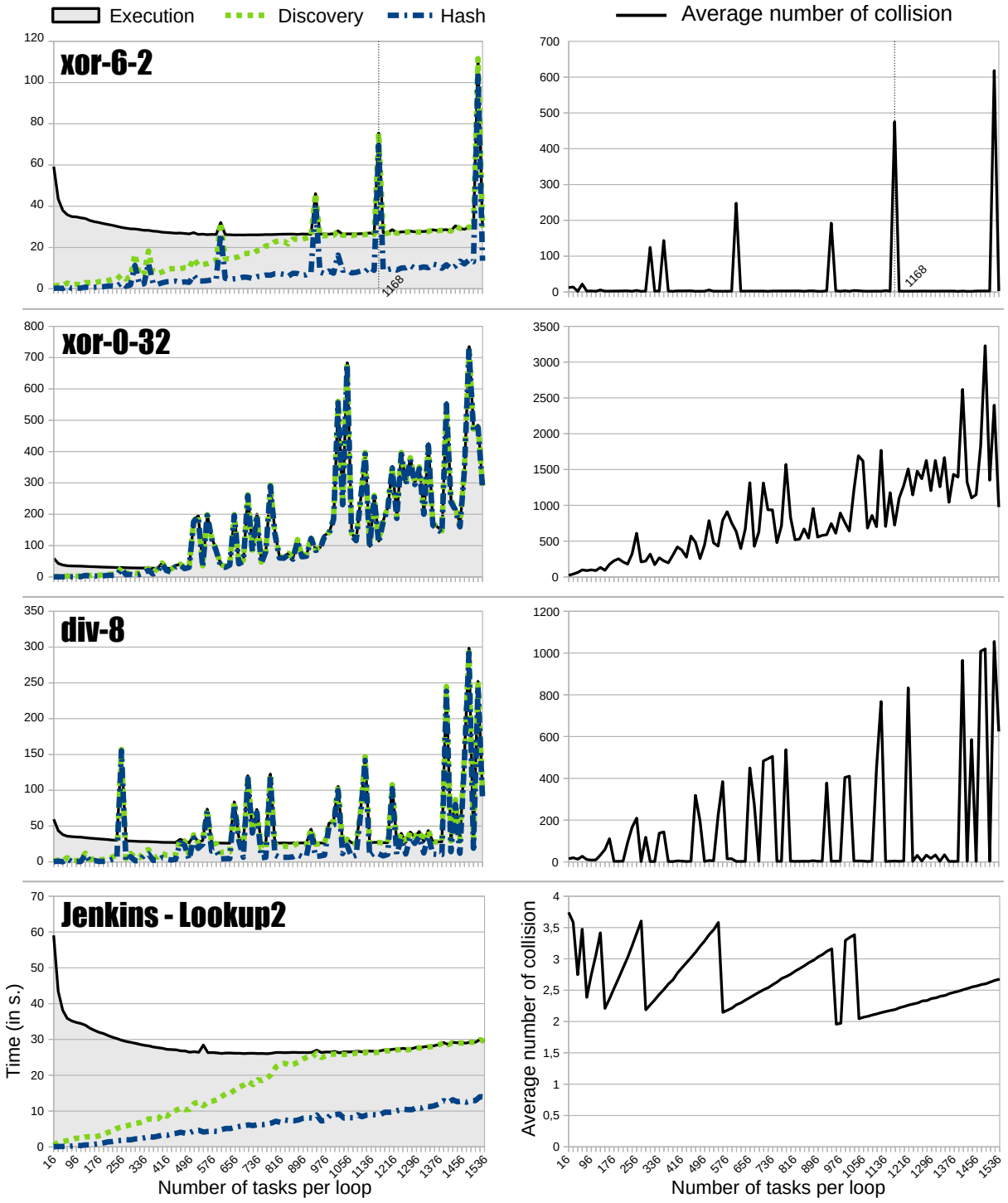


Figure 6.6: Execution and Discovery time of the task dependency graph of LULESH, and the number of collisions in the dependency hash map, with -s 256 -i 16 varying tel=tnl parameters and the hashing function, on 16 AMD EPYC 7763 Cores

6.2 Investigating The Task Dependency Graph Discovery Impacts on Performances

Our third observation on the LULESH task grain was that the execution time is almost half between coarse and fine grain. There may be multiple reasons on such gain, but in particular, fine grain-dependent tasking and depth-first scheduling following the data flow can improve the use of the hardware caches [8]. Using task-based OpenMP, task dependency graphs (TDGs) can be built following the actual application data flow. Therefore, if the data accessed by a task is fine-enough to fit into cache memory, scheduling its successors onto the same core can result in faster work execution, as cached memory is at least one order of magnitude faster to access than the DRAM. However, as the dependency graph creation (its *discovery*) is concurrent to its execution, the entire graph is never fully known by the task scheduler at run-time. In addition, refining tasks means more nodes to be processed by the runtime, and so it may slow down the discovery. Hence, refining tasks grain and accelerating the graph discovery are two critical concepts to balance: tasks must be fine-enough so that their accessed data fit into caches, but the discovery must be fast enough to provide an in-depth vision of the TDG to the runtime scheduler to benefit from predecessors cached-memory.

6.2.1 Motivations on the Dependency Graph Discovery

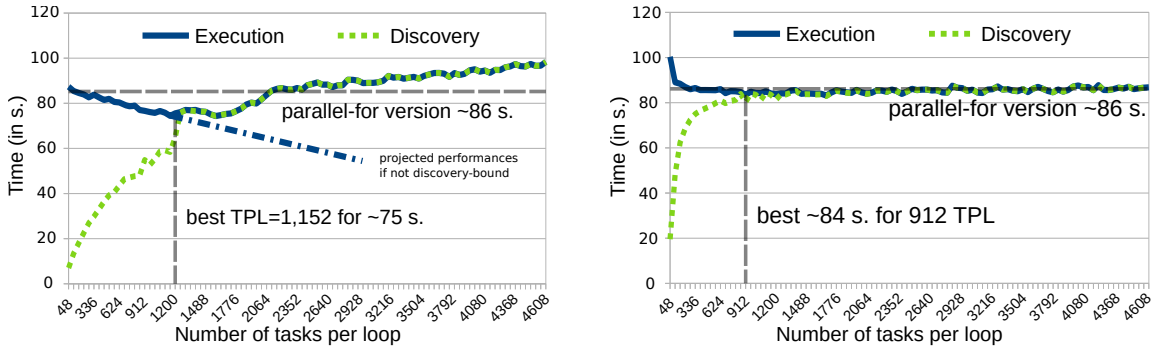
Fig. 6.7 presents our task-based version of LULESH performances using LLVM 16 and GCC 12.2.0 filling 78% of the DRAM (`-s 384 -i 16`), and with default throttling parameters. Each run is performed on a single process of 24 threads bound 1:1 onto Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz cores sharing a NUMA domain. The `parallel for` version takes about 86s to execute with 98% work time reported by Caliper [162]. From left to right on the task-based version, the number of tasks increases, and grains are refined. Figures report the time (in seconds) for the task dependency graph *execution* (blue filled curve) and its *discovery* (green dotted curve). The discovery is the time from the first to the last task creation, occurring on a single producer thread concurrently with its execution by any threads (including the producer). The execution corresponds to wall-clock time from the first task schedule to the completion of the last task. It only depends on the scheduler and the architecture: the performance increases while refining tasks, thanks to better data reuse enabled by the depth-first scheduler, until the task discovery becomes too slow and bounds the total execution time. Regardless of the number of tasks generated per loop, the performance is at most 12.8% better than the OpenMP `parallel for` version using LLVM (86s vs 75s) which is not enough relative to the effort of porting the application to the OpenMP dependent task model. GCC does not report any improvements using dependent tasks because throttling cannot be adjusted and does not allow depth-first scheduling, as too few tasks can co-exist.

6.2.2 Profiling LULESH with MPC

In order to provide finer analysis on the performances, we reproduced the previous experiment using the MPC-OMP runtime and our profiler to compute the time breakdown and collect hardware counters. Results are shown on Fig. 6.8

6.2.2.1 Overview of the results

On each graph of Fig. 6.8, the x-axis represents the number of Tasks per Loop (TPL), fixing `tel` = `tnl` = TPL. Fig. 6.8 (a) represents the overall number of tasks and edges discovered and Fig. 6.8 (b) depicts the average overheads and average work time per task (overhead and work time divided by the number of tasks). As shown on Fig. 6.8 (a,b), the right-most point reaches a total number of 7,5M OpenMP tasks with an average grain of 250μs. Fig. 6.8 (c) depicts the time



(a) LLVM release/16.x (b) GCC 12.2.0

Figure 6.7: LULESH performances on 24 Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz.

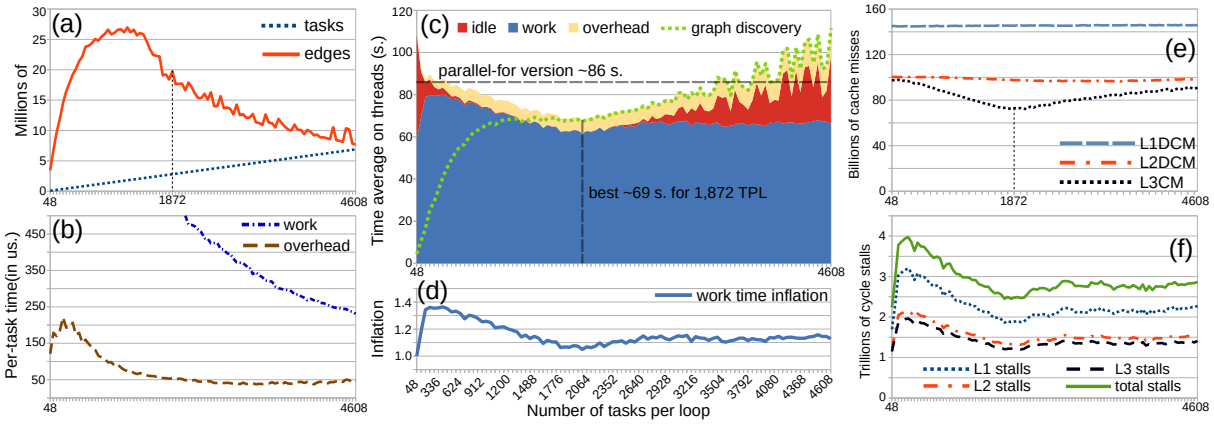


Figure 6.8: LULESH performances on 24 Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz using MPC with -s 384 -i 16 -tel TPL -tnl TPL

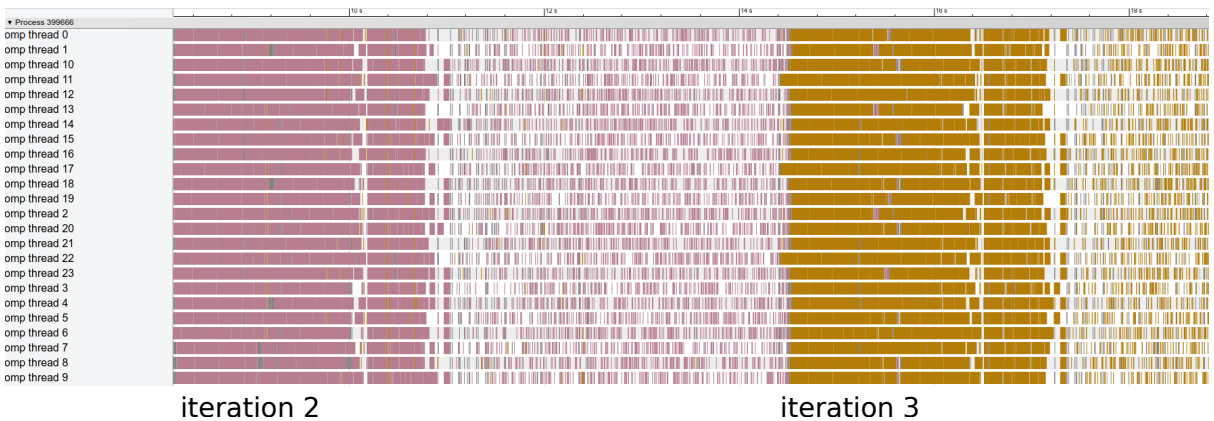


Figure 6.9: LULESH Gantt Chart on 24 Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz using MPC with -s 384 -i 16 -tel 48 -tnl 48 (left-most point of Fig. 6.8). Distinct colors indicate tasks from distinct iterations.

breakdown averaged on threads. The bottom blue stack is the work time, the middle red stack is the idle time, and the top yellow stack is the overhead. The green dotted line represents the TDG discovery time on the single producer thread. Fig. 6.8 (d) shows the work time inflation of each TPL instance using the TPL instance with the shortest work time as a reference. Fig. 6.8 (e,f) respectively depicts the number of misses and stalled cycles per cache level occurring when cores execute task work.

6.2.2.2 Interpretation of the results

Coarse grain We observe that the left-most point is the less inflated. Our interpretation comes from LULESH being memory-bound by the DRAM bandwidth: having only 48 tasks per loop reduces the amount of parallelism, which increases idleness on cores, as it can be seen on the time breakdown Fig. 6.8 (c). It can also be seen on the gantt chart Fig. 6.9, which shows iterations 2 and 3 of the left-most point (for TPL=48). In average, cores spends more time idling reducing the contention on the DRAM as less cores are likely to access the DRAM in parallel, which ends up accelerating accesses of the few actual cores working.

Middle grain From 192 to 1,872 TPL, on Fig. 6.8 (d), the work time deflates significantly from 40% inflation to 10%. Fig. 6.8 (e,f) provide an explanation: on this TPL range, we observe a reduction of L3 cache misses (L3CM) and stalled cycles due to cache-miss. Whenever a data access causes an L1DCM or an L2DCM, the memory controller will more likely find the data in the L3 memory without having to retrieve it from the DRAM. It accelerates memory access with fewer process stalls, hence the observed work time deflation. This better use of the memory hierarchy is the result of two tasking mechanisms: (1) task refinement reduces average data size accessed per task, and (2) the depth-first scheduling heuristic favors the execution of tasks' successors, improving cached-data reusability.

Fine grain After 1,872 TPL, the application execution time is bound by the TDG discovery time. Many edges are pruned because tasks are getting consummed as soon as they are produced and no longer exists on their successors' discovery, as it can be seen on Fig. 6.8 (a) This is also reflected by an increase in the idle time on Fig. 6.8 (c) where threads ends up idling waiting for tasks to spawn. Being *discovery-bound* limits the vision of the TDG to the OpenMP scheduler: the depth-first scheduling heuristic cannot be effective because successors are not known by the scheduler on predecessors completion. Threads end up scheduling whatever is ready (in a *breadth first* manner) resulting in poor cache re-usability.

Instance	Idle (s.)	Work (s.)	L2DCM	L3CM
1,872 TPL (concurrent)	3.5	1,477	98B	73B
4,608 TPL (concurrent)	766.6	1,589	98B	91B
4,608 TPL (non-concurrent)	1.2	1,077	84B	53B

Table 6.3: Impact of the task graph discovery on the work time

6.2.2.3 Impact of the task dependency graph discovery on the work time

Overlapping TDG discovery with its execution has an impact on the task scheduler that promotes data reuse: if the TDG discovery is too slow, the data-producing task will never activate its successor as it has not been discovered yet. To measure the impact of the TDG discovery on its execution time, we run a complementary experiment blocking execution by threads until the TDG has been fully discovered so that the MPC-OMP scheduler has access to the description of all the dependencies before making decisions.

Table 6.3 shows the cumulated work and idle time on threads, the number of L2 data cache misses, and L3 cache misses on different configurations. Instances with tag *concurrent* correspond to execution with TDG concurrent execution and discovery of the TDG. The Instance with tag *non-concurrent* means the graph is first fully discovered before starting its parallel execution. The two first rows present results from the best (1,872 TPL) and finest (4,608 TPL) grain execution shown on Fig. 6.8, under the concurrent configuration. The third row presents results with 4,608 TPL with the non-concurrent configuration. Comparing the two configurations with 4,608 TPL, in-depth knowledge of the TDG leads to significantly reducing the cache misses in L2 (-15%) and L3 (-42%) for a 32% work time reduction. We also observe almost no idleness, as threads do not have to wait for tasks to spawn. Work time and idleness gain lead to a 40% parallel execution time reduction. Though, the total execution time is still much slower in the non-overlapped configuration because the entire graph must be unrolled sequentially first: $357s$ with the non-overlapped experiment and $112s$ for the normal execution.

6.2.2.4 Summary

In this section, we presented an analysis of the gains from a task-based LULESH against the `parallel for` version. The task-based version executes in $69s$ with MPC-OMP against $86s$ for the `parallel for` version on LLVM 16. We measure a minimal time and work time at TPL=1,872, showing the importance of refining tasks until the TDG discovery speed becomes too limiting. Moreover, we also report that accelerating the TDG discovery on the task-based version would:

- allows reaching finer task grain so accessed data fits into the L2 cache level, which could reduce work time by 37% with effective depth-first scheduling.
- slightly reduces idleness as more parallelism is discovered.

Therefore, in the following section, we propose various techniques and evaluations for accelerating the TDG discovery.

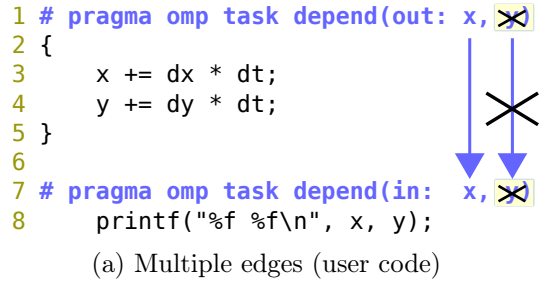
6.2.3 Accelerating the Dependency Graph Discovery

As the TDG discovery speed is a limiting factor to reach high performances, we explored way to accelerating it. In this section, we present OpenMP TDG discovery optimizations on OpenMP on runtime and user codes. Some of them are standard, but we also propose an extension caching internal tasks data structure (or task *persistence*) with very lite intrusive cost in terms of user code modification. Altogether, optimizations accelerate the TDG discovery, and we report their individual impact on performances in the evaluation Table 6.4.

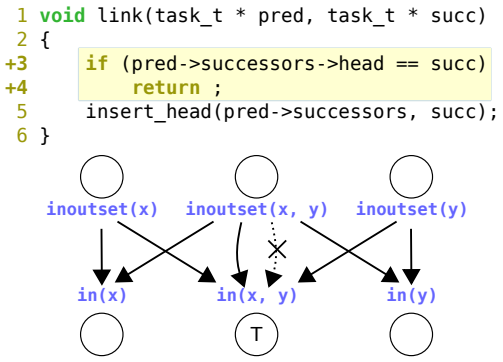
6.2.3.1 Reducing the number of edges

In the TDG, an edge corresponds to a precedence constraint between two tasks. Because of the sequential task submission and the very local view (per-procedure) of tasks, some unnecessary edges may appear between tasks which the correct order of execution is already guaranteed from other edges. Removing these edges can accelerate the TDG discovery: we present three optimizations in this direction.

The optimization (a) is depicted in Fig. 6.10a. It consists in minimizing the number of dependencies expressed from the *user code*, preserving both the parallelism and the correct order of execution. In this example, task line 1 writes (x, y), which is only read by task line 8. This provides two data addresses to be processed by the runtime while only one is really needed. Such dependency pattern appeared in our original porting of LULESH in [22].



(a) Multiple edges (user code)



(b) Multiple edges (runtime)

Figure 6.10: Optimizations (a) and (b)

Fig. 6.10b presents the optimization (b). It consists in suppressing multiple dependency edges between the two same tasks automatically by the runtime. The function `link` corresponds to the runtime procedure whenever an edge is created from the task `pred` to `succ`, by inserting `succ` to `pred` successors' list. Because of the sequential task flow semantic of OpenMP, multiple edges can be detected with a simple O(1) comparison, as if multiple edges are being generated, then `succ` is necessarily the last task inserted in `pred` successors' list. The highlighted code was added to the runtime, and perform such checking before inserting it. Note that optimization (a) differs from (b) in the sense that it not only merges multiple edges but also the cost of processing and detecting them in (b). This optimization is implemented into GCC 12.2.0 but not into LLVM 16: we implemented it in MPC-OMP.

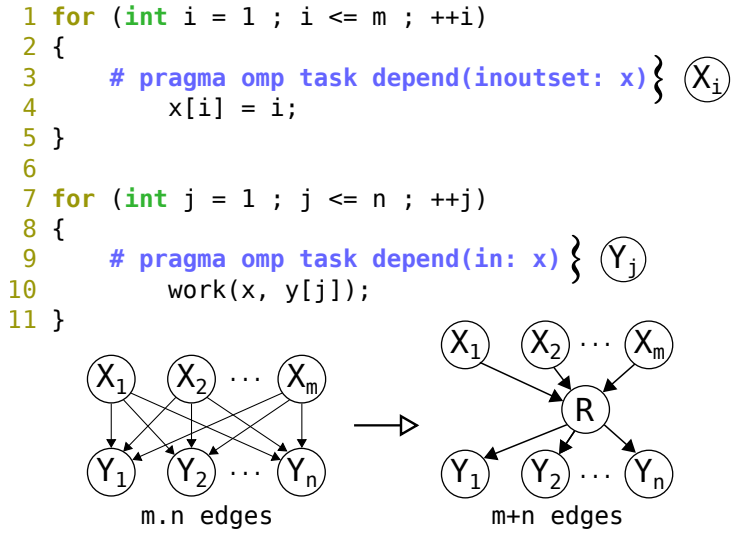


Figure 6.11: Optimization (c) - inoutset edges reduction

The optimization (c) is related to the `inoutset` dependency type. Tasks t_i having a depend clause of type `inoutset` on the same data can run concurrently, but successor tasks with any other dependency type on the same data will depend on every t_i task. This dependency type is also known as *concurrent write* in Athapascan [61], Kaapi [60] or OmpSs [86], and expresses that multiple tasks can write concurrently in a memory block. This knowledge can be used by the runtime for optimizations on the TDG edges. Fig. 6.11 depicts a minimal example of an `inoutset` dependency scheme where m tasks $(X_i)_{i\in[1,m]}$ concurrently write onto the vector x which is read by the n tasks $(Y_j)_{j\in[1,n]}$. The optimization (c) consists in inserting an extra empty

control-flow node (R) by the producer thread on the first `inoutset` for a given data, reducing the number of edges from $m.n$ to $m+n$. This optimization had been implemented in LLVM [4] but not into GCC: we implemented it in MPC-OMP.

```
1  # pragma omp persistent
2  for (int iter = 0 ; iter < iter_max ; ++iter)
3  {
4      # pragma omp taskloop firstprivate(iter) num_tasks(t)
5          depend(in:  dt) depend(out:  mesh.nodes[i])
6          for (int i = 0 ; i < mesh.nodes.size() ; ++i)
7              work_on_nodes(mesh.nodes[i], dt);
8
9      # pragma omp taskloop firstprivate(iter) num_tasks(tel)
10         depend(in:  dt, mesh.nodes[...])  depend(out:  mesh.elements[i])
11         for (int i = 0 ; i < mesh.elements.size() ; ++i)
12             work_on_elements(mesh.elements[i], dt);
13 }
```

Listing 6.10: Persistent Task Graph - API

```
1  mpc_omp_persistent_region_begin();
2  for (int iter = 0 ; iter < iter_max ; ++iter)
3  {
4      mpc_omp_persistent_region_iterate();
5      # pragma omp taskloop firstprivate(iter) num_tasks(t)
6          depend(in:  dt) depend(out:  mesh.nodes[i])
7          for (int i = 0 ; i < mesh.nodes.size() ; ++i)
8              work_on_nodes(mesh.nodes[i], dt);
9
10     # pragma omp taskloop firstprivate(iter) num_tasks(tel)
11         depend(in:  dt, mesh.nodes[...])  depend(out:  mesh.elements[i])
12         for (int i = 0 ; i < mesh.elements.size() ; ++i)
13             work_on_elements(mesh.elements[i], dt);
14 }
15 mpc_omp_persistent_region_end();
```

Listing 6.11: Persistent Task Graph - Runtime calls

6.2.3.2 Persistent Task Dependency Graph

Scientific simulations often are iterative applications with similar existing parallelism over iterations. For instance, LULESH is a simulation on an unstructured mesh, and the task dependencies are built upon the mesh nodes and elements that do not change over iterations. To accelerate the TDG discovery furthermore, we propose an original interface and runtime implementation that adds persistence to tasks and edges. Looking back at OpenMP tasks transition system (see Fig. 3.2), persistence means adding a transition (8) → (1).

Implementation The optimization (p) is illustrated on Listing 6.10, and from a programmer's point of view, it only consists in annotating a loop generating dependent tasks in the same order and with the same dependency scheme on each iteration (line 1). Listing 6.11 shows the code generated by the pragma. Line 1, the `mpc_omp_persistent_region_begin` initializes a persistent region. Then on the first loop iteration, the application TDG is discovered just as before, but:

- Marking tasks as *persistent*, so they are not destroyed on completion,

[4] `https://reviews.llvm.org/D97085`

- Putting tasks in a list L sorted by their discovery order.
- Creating every edge, as opposed to the non-persistent mode, which would prune edges from tasks already consumed: since no edges are created on the next iterations, generating every edge ensures the correct order of execution with no race conditions on the discovery. It also makes the first iteration of persistent TDG discovery slightly more costly than its non-persistent version.

In every iteration, the `mpc_omp_persistent_region_iterate` waits for the completion of every persistent task previously instantiated, and moves a pointer p to the beginning of L. On the next iterations, the producer thread executes the same execution flow, but on a task construct, it only:

- Retrieves the persistent task t pointed by p and increment p,
- Copies t `firstprivate` data (such as the loop `iter` variable),
- Transition t to a `queuable` state,

Persistence relieves the runtime from other tasking overhead sources such as the internal task descriptor allocation, dependencies processing (`depend` clause), or Internal Control Variable (ICV) management. Therefore, a task initialization cost is reduced to a single `memcpy` on `firstprivate` data, representing 8 to 100 bytes in LULESH tasks. An implicit barrier at the beginning of each iteration ensures that every task is completed before re-instancing them.

Remark Coupling optimization (c) with (p) required a specific treatment by the runtime, as the control-flow tasks inserted by (c) are not only discovered by the producer thread once when processing dependencies on the first iteration. Instead, control-flow tasks are re-instantiated by the first consumer thread that executes one of its `inoutset` predecessors of the current iteration.

Applicability A persistent task graph assumes that dependencies between tasks remain constant over iterations. However, on simulation over unstructured meshes like LULESH, Adaptive Mesh Refinement (AMR) may occur at some point, slightly changing the mesh. Nevertheless, because of the inherent costs of mesh adaptation, simulations try to amortize it over a few iterations. The persistent task graph could be invalidated when AMR occurs so that the producer thread re-construct is entirely on the next iteration. Such an approach would allow fast integration of persistent task graphs into existing applications with this already existing strategy every few iterations.

On the other hand, AMR only slightly changes the mesh, and it may be unnecessary to re-build the entire graph, as only a small portion of the nodes and edges may have changed. Removing a persistent task from the TDG is rather straightforward: We even added an `mpc_omp_persistent_region_pop_task` API that deletes the task currently pointed by the producer thread and increments the pointer to the next task. However, inserting a new task is more complex and was not studied furthermore as part of this thesis.

Impact on user codes and runtimes The impact of persistent TDG on user code is lite: from given task-based applications, a single line of code (LoC) annotation on a loop enabled persistence. Our implementation moderately impacted MPC-OMP runtime, adding only 175 LoC but changing critical and highly concurrent runtime code related to tasks' life-cycle.

6.2.3.3 Evaluation

Table 6.4 depicts the number of edges, the TDG discovery, the total execution time, and the work/idle/overhead time breakdown, crossing each optimization on the same problem as of Fig. 6.8 with $TPL = 1,872$; for about 2.9 M. tasks of $365\mu s$ grain in average. We make the follow interpretations comparing pairs of lines:

optimizations	n° of tasks	n° of edges	discovery	execution	work	idle	overhead
none	2,802,400	56,320,296	76.67	77.48	1,612	3.49	229.50
(a)	2,802,400	58,168,382	71.79	73.40	1,533	3.00	211.40
(b)	2,802,400	21,926,084	67.53	71.83	1,546	2.85	131.70
(c)	4,853,811	30,246,767	71.91	73.19	1,568	3.07	167.44
(a)+(b)	2,802,400	31,144,779	66.60	69.39	1,503	3.23	144.26
(a)+(c)	3,614,124	36,239,253	56.38	69.43	1,512	3.31	128.51
(b)+(c)	4,858,307	24,837,165	71.73	72.95	1,568	2.85	163.29
(a)+(b)+(c)	3,613,737	27,633,254	55.56	69.17	1,517	2.64	116.77
(a)+(b)+(c)+(p)	3,557,840	19,177,294	5.97	70.80	1,655	12.22	27.30

Table 6.4: Optimizations crossing on Intel Skylake nodes LULESH -s 384 -s 16 -TPL 1872. Times are given in seconds

- (b) against (a)+(b) - more edges are generated in the second case even though TDGs discovery are about the same. This rebound-effect phenomenon occurs because the TDG discovery is faster with optimization enabled, leading to more predecessors existing on successors' creation; therefore, less automatic pruning occurs. The same phenomenon occurs comparing *none* against (a).
- (a) against (a)+(c) - about $800,000$ more tasks are generated, corresponding to the empty tasks of the optimization (c). However, they dampen the number of edges ($n.m$ against $n + m$ as shown on Fig. 6.11) for about $22,000,000$ fewer edges. In this configuration, the execution is no longer discovery-bound, meaning that the scheduler likely manages to perform effective depth-first scheduling.
- *none* against (a)+(b)+(c) - enabling all optimization but (p) led to 2.04 fewer edges and 1.38 speedup on the discovery over the non-optimized version.
- (a)+(b)+(c) and (a)+(b)+(c)+(p) - enabling tasks' persistence optimization (p) drastically reduces the discovery time by an order of magnitude. Among the $5.97s$ of discovery, $4.11s$ corresponds to the first iteration discovery, and the remaining corresponds to the 15 other iterations taking $0.12s$ on average: the first iteration is about 34 times slower than the others, as it is the one responsible for building the dependency graph while the others simply update tasks private data. The first persistent iteration ($4.11s$) is more costly than non-persistent iterations ($3.47s$ on average), and even though every edge is created (the runtime does not prune edges to tasks already consumed with persistent tasking enabled), we observe fewer edges enabling (p). This is a side effect of our runtime persistence implementation, which has an implicit barrier on each iteration, removing inter-iteration edges. Moreover, this barrier leads optimization (p) to increase the total execution time ($69.17s$ to $70.80s$): the work and idle times increase because the barrier prohibits successor tasks of iterations $n + 1$ from starting until every task of iteration n is completed. This is shown on the Gantt chart Fig. 6.14 where one color represents one iteration. Threads spend more time idling in this synchronization barrier, and the work time inflates as depth-first scheduling is constrained per iteration. However, as (p) accelerates the graph discovery, overheads are reduced, enabling effective depth-first scheduling at a much finer grain.

Fig. 6.12 shows the same metrics as in Fig. 6.8 but enabling every optimization. The TDG execution is no longer bound by its discovery and enables effective depth-first scheduling. It leads to 1.56x speedup over the `parallel for` version and 1.27x speedup over the non-optimized task-based version. The 4,608 TPL configuration reaches a $1,230s$ work time for $82B$ L2DCM and $54B$ L3CM.

METG report The Minimum Effective Task Granularity metric was proposed [163] as a way to assess the overheads of tasking runtimes. For a given application and runtime system,

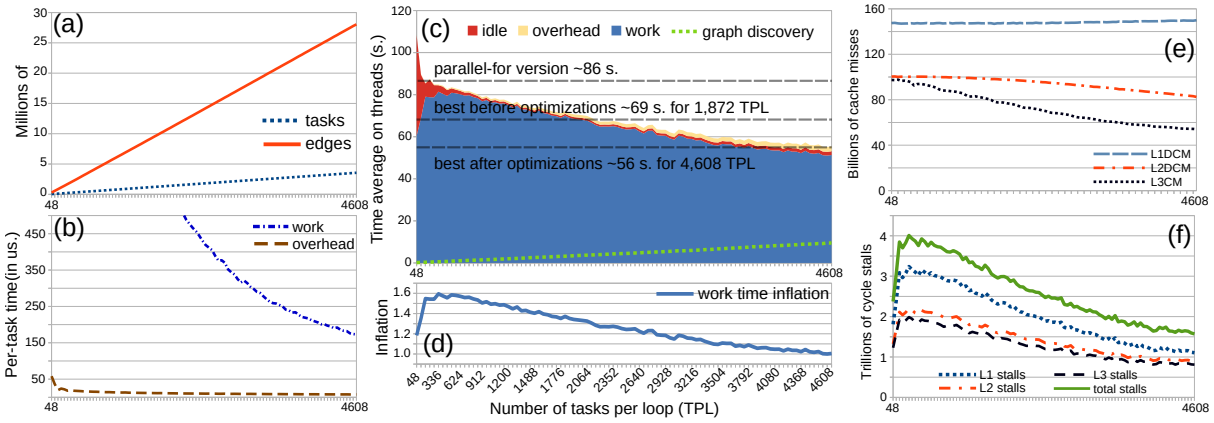


Figure 6.12: LULESH on Intel Skylake node with optimizations

$METG(X\%)$ gives the minimum task grains for which an instance of execution reaches $X\%$ of the best performances measured on any runtime system. Their results on GCC/LLVM show a $METG(95\%) = 1ms$ for several applications. Running LULESH with GCC, LLVM, and MPC-OMP tasking runtime, we measured an $METG(95\%)$ of $65\mu s$ with 9,216 TPL using MPC-OMP, which is 1.5 order of magnitude less than the best OpenMP METG reported in [163] for such efficiency.

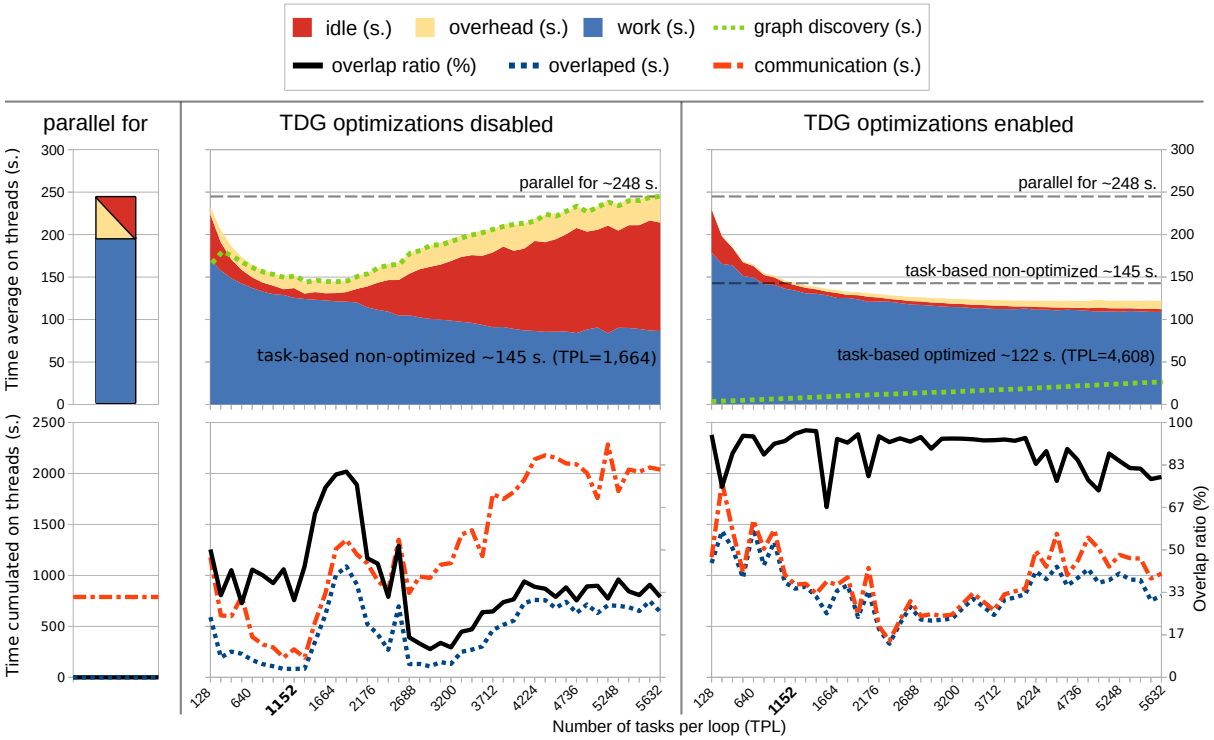


Figure 6.13: LULESH -s 256 -i 64 on 125 MPI processes, 54 AMD EPYC 7763. Time breakdown (top) and communication performances (bottom) on a profiled MPI process (rank 82)

6.2.4 Impacts on Distributed Execution

The TDG discovery speed also impacts distributed applications of applications using MPI. We evaluate three applications (LULESH, HPCG, Cholesky) with different parallel characteristics (computation, communication) that will lead to three conclusions. We performed evaluations

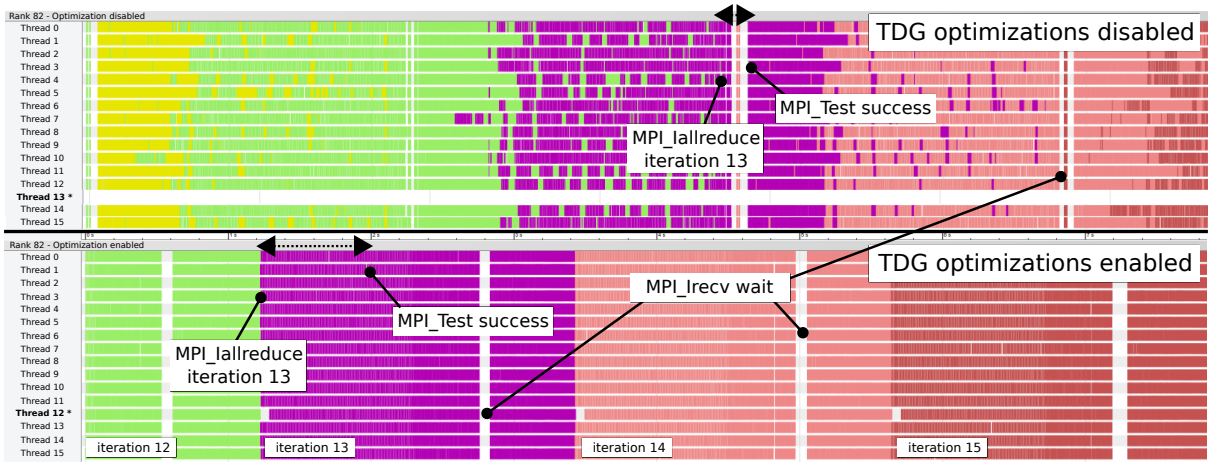


Figure 6.14: Gantt chart of task-based execution for TPL=1,152

on AMD EPYC 7763 64-core CPU, interconnected with Atos BXI V2 running Open MPI 4.1.4. In our experiment, we place MPI processes per NUMA domain with 16 OpenMP threads compactly bound 1:1 to cores. LULESH was scaled to fill 72% of each NUMA domain on this new architecture (`-s 256`), running on 16 nodes with 125 MPI processes.

6.2.4.1 Distributed Execution Performances

LULESH comes with 3 types of point-to-point (P2P) requests communicating mesh partite frontiers with neighbors: a single node, an edge, or a face, respectively, with O(1), O(s) and O(s^2) bytes transfers. For our given problem size and our Open MPI configuration, O(1) and O(s) bytes MPI requests are performed with an eager protocol, but O(s^2) requests are performed using rendezvous. As opposed to the `parallel for` version, which implies the entire mesh-wide O(n^3) computation to complete before starting communications, the task-based version allows posting of MPI requests as soon as predecessor tasks working on frontier nodes have been completed. Hence, a depth-first scheduling strategy can lead to earlier communication posting and preserve independent work for overlapping communication, which is the subject of this study. Fig. 6.13 presents performances on an MPI process connected whose mesh partite is connected to 26 other MPI processes. It shows the time breakdown and the communication time for the `parallel for` version (LLVM 16), the non-optimized task-based version (MPC-OMP), and the optimized task-based version (MPC-OMP).

Computational Performances Times were retrieved using our profiler for MPC-OMP and Caliper for LLVM (which only provides work/non-work times). We observe the same performance gain as on the previous architecture. The optimized task-based is 2.0x and 1.2x more performant than the `parallel for` and the non-optimized task-based version, for the exact same reason of hierarchical memory accesses. Note that we observe work time deflation on the non-optimized task-based version after 2,176 TPL. Above this threshold, the TDG discovery becomes too slow to feed every core (as shown by the important idleness), reducing the DRAM contention, which in turn, accelerates memory accesses of the few threads working in parallel.

Communication Overlap with Computation The `parallel for` version exhibits no overlap potential on P2P send communications. All the requests are posted in non-blocking mode, and the execution flow waits for every completion before pursuing any computation. The collective is in blocking mode, posted at the beginning of a new iteration. The collective of iteration $n+1$ depends on every task of iteration n. However, the Gantt chart of the task-based version on Fig. 6.14 shows that there are some independent computational tasks of iteration $n+1$ such as

`CalcFBHourglassForceForElems` ready for overlap. On the `parallel for` version, all this work would have been processed synchronously after the collective operation completion following the sequential instruction flow. On the task-based versions, the overlap ratio is improved when enabling optimizations and remains above 80% on any TPL against 50% on average without optimizations as shown Fig. 6.13: the application parallelism is discovered faster and more work is ready for overlap masking communication costs.

To estimate the gain from integrating MPI communications into OpenMP TDG following the data flow, we added explicit `# pragma omp taskwait` before and after communication sequences. This way, we reproduce the original application BSP communication pattern: every communications are posted once the entire domain is computing, meaning there is no computation/communication overlap nor early-bird posting. With the tracing disabled and on the most performant configuration (TPL=4,608), we measured 131.0s with `taskwait` against 121.9s with no `taskwait`, meaning that fine integration of MPI communications into OpenMP TDG reduced by 7% the total execution time.

On every iteration, with or without optimizations, we observe an idleness period on every thread on the Gantt chart Fig. 6.14. During this idle period, a single `MPI_Irecv` is blocking, and no more work is ready until its data are received. This behavior is similar to what we observed on the Cholesky factorization in Section 5.2. Therefore, we enabled our heuristic favoring send operations and evaluated for $TPL = 1,152$, hoping that the `MPI_Recv` idleness observed on the Gantt chart may be removed due to earlier data posting remotely. Nevertheless, preliminary results have shown no improvement nor degradation on any of the work/idle/overhead metrics, likely because the default depth-first scheduling heuristic already lead to early-bird communication anyway.

Communication Time We made a breakdown of the communication time of each TPL, and on average, 94% of the communication time corresponds to the `MPI_Iallreduce` collective, and the remaining 6% are the 26 P2P send operations. The variation observed in the communication is, therefore, mostly related to this single collective.

Gantt charts Fig. 6.14 shows the execution of tasks from iterations 11 to 15 on the two task-based versions of $TPL = 1,152$ of Fig. 6.13. Boxes represent task schedules, and each color identifies a different iteration. Time origins had been offset on each chart along the x-axis to align with the first task of iteration 12 (green). Because of the implicit task barrier on each iteration induced by our persistent TDG implementation, no tasks from iteration $n + 1$ can start until every task from iteration n is completed (bottom chart). Looking at the two charts, this behavior seems to inflate collective synchronization time globally. This phenomenon may explain why the (collective) communication time is faster on the non-optimized version on $128 < TPL < 1,280$. Though, it has really little impact on performances as it is getting overlapped with work anyway.

Refining from 128 to 1,280, we also observe a reduction in communication time. We believe it comes from faster local execution (as shown on the time breakdown), allowing, on average earlier collective communication matching on every MPI process.

However, refining furthermore on the non-optimized version leads to important idleness, as the TDG discovery becomes too slow to feed every core with work. A possible explanation is that every MPI process must wait for the slowest local OpenMP TDG discovery, as results show that accelerating discovery lead to lower communication time at a fine grain.

6.2.4.2 Scaling to 65k cores

We performed a strong and weak scaling on LULESH for both the `parallel for` (GCC 11.2.0) and our optimized task-based version (MPC-OMP). We increased the number of simulation iterations from 64 to 1,024. Table 6.5 presents wall clock time results scaling from 8 MPI processes (= 1 node) to 4,096 processes on a single run. No performances could be recorded on the weak-scaling above 1,331 processes because all versions abort on a numerical error.

MPI processes	8	27	64	128	216	343	512	729	1,000
weak - for (s.)	3,926	3,935	3,953	3,954	3,979	4,006	4,012	4,141	4,181
weak - task (s.)	2,065	2,136	2,074	2,088	2,153	2,077	2,093	2,185	2,089
strong - for (s.)	3,926	895	305	150	85	61	44	30	19
strong - task (s.)	2,065	627	267	148	88	69	49	43	29
strong - TPL	2,048	599	256	129	74	47	32	21	16
strong - nodes per tasks	8,192	8,202	8,192	8,226	8,298	8,276	8,192	8,362	8,290
strong - memory used	72%	21%	9%	4.6%	2.6%	1.7%	1.1%	0.8%	0.6%

MPI processes	1,331	1,728	2,197	2,744	3,375	4,096
weak - for (s.)	N/A	N/A	N/A	N/A	N/A	N/A
weak - task (s.)	N/A	N/A	N/A	N/A	N/A	N/A
strong - for (s.)	15	11	10	8	10	11
strong - task (s.)	23	16	16	11	13	9
strong - TPL	16	16	16	16	16	16
strong - nodes per tasks	6,083	4,630	3,707	2,916	2,456	2,048
strong - memory used	0.4%	0.3%	0.3%	0.2%	0.2%	0.1%

Table 6.5: LULESH -s 256 -i 1024 - Weak and Strong scaling from 8 to 1,000 MPI processes

From 8 to 1,000 MPI processes with 2,048 TPL, task-based executions lasted about 2,000 s. with more than 95% weak-scaling efficiency and a 2.0x speedup compared to the `parallel for` version. The strong scaling from 8 to 4,096 MPI processes uses a dynamic TPL to balance parallelism and workload per task, ensuring at least 16 tasks per loop and, at most 8,192 mesh nodes per task, as shown in the last row. The task-based version improves performances over the `parallel for` version until 128 MPI processes for about 5% DRAM use.

6.2.4.3 HPCCG

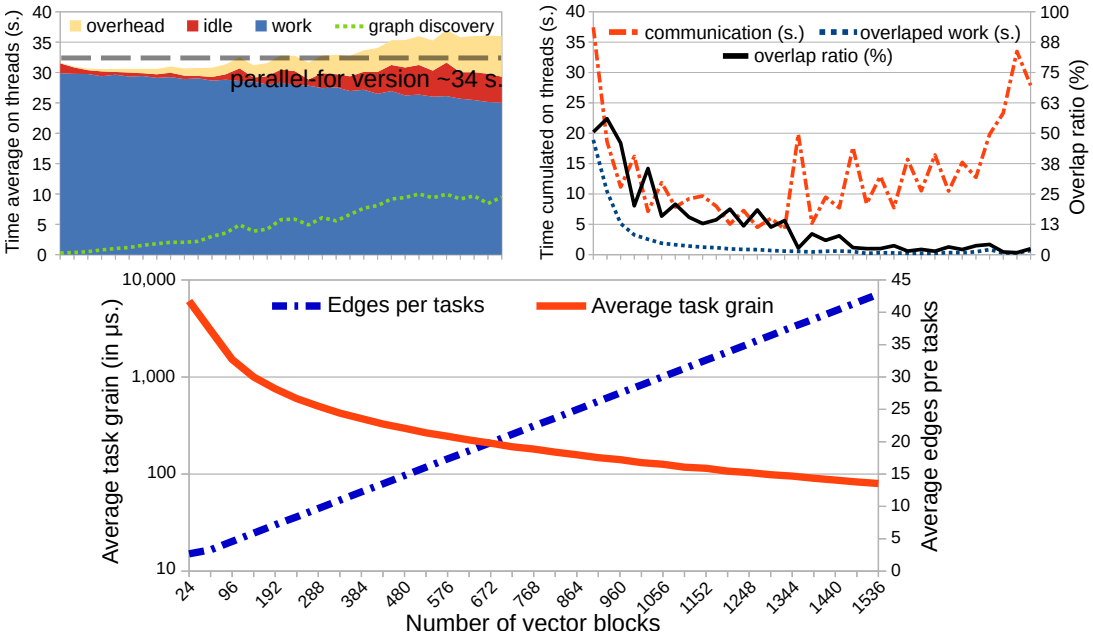


Figure 6.15: HPCCG performances on 32 MPI processes on Intel processors for a matrix with n=41,943,040 on i=128 iterations

The High-Performance Conjugate Gradient (HPCCG) is a benchmark used to rank supercomputers as a complement to the LINPACK (HPL) benchmarks [148]. The baseline implementation is parallelized using `parallel for` construct and barriers before communicating with MPI.

Section 6.1.1 presented the porting of HPCG to OpenMP dependent task model with two grains parameters defining the number of block for vector-wise operations (TPL) and the number of sub-blocks for SpMV operations, that we fix to 32 in future experiments. Fig. 6.15 present our results on 32 MPI processes of 24 threads varying the TPL parameters on Intel Skylake processors.

On Communications Depending on the TPL, the communication time varies from 5 to 37 s. for an accumulated work time of around 700s. It means with perfect overlap, at most 5% of the work time would be overlapping communication. Moreover, even though TDG discovery is fast-enough, the overlap ratio remains low ($\leq 23\%$), meaning little work is available in parallel of communications. Therefore, there is little no to gain to expect from overlapping communication with computational tasks on HPCG.

Time Breakdown Regarding work time, the best performances are reached for an average tasks grain of $80\mu s$ using the right-most 1,536 TPL. It allows 20% work time reduction compared to the baseline `parallel for` version. The reasons are the same as LULESH: memory accesses are faster due to better cache reuse. Even though TDG discovery optimizations were enabled, fine-grain tasks management overheads deteriorated total execution time more than the work time gain. Though, the minimal total time ($30.6s$) is obtained with TPL=144 ($1ms$/tasks) for 10% performances gain over the `parallel for` version with LLVM 16 ($34.1s$. with 95% work time). Above this grain, overheads, and idleness deteriorates more performances than the work time improvement. We explain this by the *tasking runtime contention*: as shown in Fig. 6.15, the number of edges per task grows linearly from 24 to 1,536 TPL while the workload per task decreases. Runtime-side, this is reflected by more threads accessing more often shared data structures, such as the task dependency graph.

6.2.4.4 Tile-based Cholesky Factorization

Tile-based Cholesky dense matrix factorization is a widely studied application of dependent task-based programming. We retrieved the version of [99] with dependent task and MPI communications performed by tasks. Optimizations (a), (b), and (c) does not provide any performance improvement/degradation as they are not reflected in the application: its dense dependency scheme is simpler than sparse and indirection-based data structures found in HPCG and LULESH. The optimization (p) provides performance gain on TDG when iteratively decomposing matrices of the same dimensions and tile size. We evaluated the TDG discovery speed gain for a matrix of size $n = 65,536$ with block sizes $b = 512$, distributed on 32 MPI processes of 24 cores over 16 Intel nodes. Our results showed 5x asymptotic[5] speedup on the discovery when increasing the number of iterations. It showed no significant impact on performances as the TDG discovery is already fast due to coarse tasks and regular dependencies (<2% of total time). On 16 Skylake nodes (768 cores), we measured 269s (with) and 274s (without) optimizations for matrix n=65,536 with block-size m=512.

6.2.4.5 Summary

In this section, we combined optimizations to hasten the TDG discovery. Optimization (a) is well-known, (b) is implemented in GCC, (c) is implemented concurrently to us in LLVM, and (p) is original. One contribution lies in combining them and evaluating their gain.

Then, through the LULESH case study, we analyzed the impact of the TDG discovery on performances. Our best result shows a 2.0x speedup on LULESH using OpenMP tasking over

[5] the asymptotic speedup corresponds to $\frac{t_p}{t_{np}}$ with t_p and t_{np} respectively the discovery time of a persistent iteration, and a non-persistent iteration

`parallel for` weak-scaled to 16,000 cores, thanks to task grain refinement and effective depth-first scheduling, that were only possible by setting up a suitable task throttling and hastening the TDG discovery. Finely composing MPI and OpenMP by integrating communications into dependent tasks led to 7% execution time reduction. HPCG evaluations report moderate performances with a 1.1x speedup due to difficulties in balancing work time gains with runtime contention. Cholesky results do not provide either performance improvements or degradation.

6.3 Related Works

Persistent Task Graph In 2021, C. Yu et al. [164] proposed an extension to the OpenMP specifications for *taskgraph* to capture the graph unrolled by a code section in order to re-execute it. Their proposal slightly differs from ours, as all the closures are only captured in the first iteration with their taskgraph: the producer thread execution flow is not re-executed on next iterations, so the `firstprivate` data may not be updated. Moreover, they only report provides a standard interface simulated results, while we also provide a runtime implementation and experimental results. In late 2022, the same authors uploaded complementary work on arVix [165] with a runtime implementation and speedup evaluations on shared-memory microbenchmarks. As opposed to their approach, our producer thread re-executes the instruction flow on every iteration. It can update task parameters (firstprivate data, if clause, ...) which preserves partially taskified programs conformity; authors conclude that it is a limit of their approach. Additionally, we evaluated an irregularly distributed proxy application (LULESH) reporting metrics; and showing impacts of/on the TDG discovery that are ignored on the arVix paper (cache re-use, communication overlap, throttling): they only report speedups on single-node Cholesky/Heat/Microbenchmark that may be inflated due to arbitrary fine-grained tasks. Nevertheless, their compile-time proposal nicely complements our runtime implementation.

In 2023, D. Alvarez et al. [166] proposed a cyclic data-flow task graph interface and implementation for iterative applications. Their approach is semantically identical to our persistent task graph: marking a loop generating the same TDG on every iteration, with the possibility to replay the producer task execution flow to update `firstprivate` data. However, as opposed to our runtime implementation, theirs does not have an implicit barrier on every iteration. Instead, the correct order of execution is guaranteed through a *cyclic* graph. To do so, their runtime stores multiple versions for each task and keep track of the TDG roots and leaves to insert cyclic edges. The detailed analysis and results we presented in this thesis on LULESH motivate the removal of our persistent task graph barrier on every iteration, as it limits scheduling possibilities and leads to work time inflation. Yet, remains to be evaluated if cyclic graph additional costs (multiple instances on-fly for the same task and leaves/roots tracking) will not deteriorate performances more than the gain obtained from the barrier removal. Likely, there is no fix answer on this, and an interface flexible-enough to have control on the iteration barrier could make sense.

Optimizations in Tasking Runtimes The OpenMP dependent task model was adopted since the specifications 4.0 [90] but implementations induces important overhead [7]. The history of optimization in task graph runtime starts at least twenty years before OpenMP 4.0. Cilk [87] was the precursor by considering end-to-end optimizations from the compilation a fast and slow versions of each task to the implementation of the stealing protocol without costly lock operation on the general case execution path. This was the starting point to several paper related to work stealing scheduler optimization which is outside the scope of our TDG optimization. Parsec [167] based their graph model on the parametrized graph model [168] to implicitly represent tasks and their dependencies. It was a huge step in optimizing the memory space related to a task graph. Nevertheless, because this model does not fit well with irregular applications, the authors have also added a the classical task graph model proposed since the mid-90s [54, 60, 61, 86] built at run-time from description of tasks and their accesses (read,write) to the memory. Task graph

optimizations presented in this chapter are almost present in all these runtime, but our runtime is the first OpenMP runtime that systematically integrates them.

6.4 Conclusion

The lack of applications using the task-based composition of MPI and OpenMP lead us to port a benchmark (HPCCG) and a proxy-application (LULESH) ourselves. During our porting journey, we faced the three programming/performances/profiling difficulties that motivated this thesis, and proposed solutions.

On programming aspects, both HPCCG and LULESH present irregular accesses which are hard to express under the current OpenMP specifications 5.2. Removing a few standard restrictions could ease future porting of applications. On profiling and performances aspects, our results on LULESH suggests that OpenMP TDGs discovery, which rely on a read-after-write (RaW) model, implies important runtime overheads with about 37% of the total discovery time on the LULESH producer thread. It makes it a critical component of task-based applications and runtimes to be profiled for assessing on performances; and optimized whenever bounding the execution time.

In our evaluation cases saturating the DRAM capacity, the TDGs discovery speed was an issue on both Intel Skylake and AMD EPYC nodes. It lead us to propose several performances optimizations: in applications code (reducing the number of dependencies expressed, and using the `inoutset` type); in the MPC-OMP runtime (redundancies filtering, `inoutset` efficient implementation, and task persistence); and in the OpenMP programming standard (task persistence). We evaluated the impact of each optimization on performances which lead us to the following conclusions:

- Fine-grain dependent tasking can improve performances of memory-bound applications whenever the domain computed does not fit into caches. Using a depth-first scheduling strategy, we showed 2x work time deflation thanks to temporal locality into caches.

- If an execution is discovery-bound, tasks may be consumed before an independent communication is discovered (and therefore, consumed), reducing the available work for overlapping.

- Whenever using task persistency (or cyclic graphs), a barrier at the end of each graph iteration may improve (by reducing overheads) or deteriorate performances (by inflating work time). If such persistence interface were to be standardized in OpenMP, programmers should probably be given a way to enable/disable this implicit barriers to match their use-cases. Further evaluations should be conducted to assess more precisely on benefits/losses of such a barrier, for instance by comparing our approach (with a barrier) to D. Alvarez et al. [166] (with no barriers).

Part III

Conclusion and Perspectives

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

Executing numerical simulations on supercomputers is a powerful tool for scientific research. To assist scientific simulation programmers, the HPC community has been developing several programming models bridging codes with the ever-complexifying hardware. In particular, the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) are two standard programming models developed by large and diverse communities. They respectively target inter-node and intra-node parallelism, so their mixed-use appears as a well-suited solution to ensure both source code and performance portability across supercomputer architectures.

OpenMP dependent task-based programming model opens the door to new ways and benefits of mixed-use, such as the implicit overlap of MPI communications with OpenMP tasks or the early-bird posting of MPI communications through the OpenMP task scheduler. However, porting existing scientific simulation codes to this new hybrid model raises the three Profiling/Programming/Performance difficulties we introduced in Chapter 1.

In this thesis, our contributions consisted of the investigation and solution proposals on these three difficulties:

- On profiling, we proposed a unified task-based performance modeling of MPI+OpenMP applications. We defined performance metrics and developed a run-time profiler and post-mortem analysis. For such mixed-use, the work/idle/overhead time breakdown and overlap metrics we profile are original, and their use throughout this thesis shows how necessary they are to assess performances. Additionally, task-based metrics (number of tasks, edges, granularity, per-task hardware counters) and visualizations (Gantt charts, dependency graphs) for such mixed-use were significantly helpful for interpreting presented results. Finally, our work on the task dependency graph showed the interleaving between its discovery and its execution (work time inflation, overlap potential reduction): it must be taken into account when profiling task-based applications.

- On programming, and to respond to the lack of applications currently taking benefits of the mixed-use using tasks, we ported two mainstream HPC applications: the High-Performance Conjugate Gradient (HPCG) and the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) providing experience feedback.

- On performances, our developments in the MPC-OMP runtime led to a cutting-edge and open-source OpenMP tasking runtime implementing the most recent features. In Chapter 6, we showed the criticality of the task dependency graph discovery and cross-evaluated several optimization techniques for its acceleration. Finally, preliminary results on our cooperative target task design suggests potential performances improvements over existing solutions at fine-grain GPU offloading.

Each of research axis contributed to propose extensions to the OpenMP specifications.

- Section 5.1 propose to differ the responsibility of tasks execution context to programmers and a new task suspension/resumption mechanism. They aim to automate the overlap and progression of external asynchronous operation through the OpenMP task scheduler.
- Section 5.2.3 extends the interpretation of the `priority` clause so it is back-propagated to predecessors. This way, programmers can guide the task scheduler towards critical tasks without the need to manually compute priorities of every tasks.
- Section 6.1.1 proposes an extension on existing the `depobj` construct and `iterator` to simplify the expression of irregular dependencies.
- Section 6.2.3.2 introduces task *persistence* to cache the task dependency graph of iterative applications, drastically reducing fine-grain tasking overheads.

This work lead to two publications in the International Workshop on OpenMP (IWOMP) [130,131] and a publication in the International Conference on Parallel Processing (ICPP) [169]. Another publication in IWOMP [22] presented preliminary experiments which was not presented as part of this document, adapting our work for GPU offloading using OpenMP `target` tasks.

Finally, even though tasking comes with management overheads over traditional work-sharing loops, our contributions illustrates it may still be beneficial to overall performances:

- *Early-bird posting* using tasks reduced idleness and slightly improved overall performances ($\sim 3\%$) on a Cholesky factorization.
- *Application-wide load balancing* and *depth-first scheduling* - significantly reduced idleness and work time on LULESH for a 2x speedup when saturating the DRAM capacity.

In the end, our contributions only provide partial responses on each of the three original difficulty, which lead us to present a few research directions to pursue our work.

7.2 Perspectives

7.2.1 Tools for Assisting Dependent Task-based Programming

We introduced a new profiler for the mixed-use of MPI and OpenMP using tasks. An important contribution was the post-mortem computation of the work/idle/overhead time breakdown heavily used for performance interpretation throughout this thesis. Still, mixed-use of the standard using tasks currently lacks such fundamental performance analysis tool accessible to a non-expert audience. We propose a few directions to explore to improve on debugging and profiling aspects.

Extending Our Profiler Firstly, our profiler would need to be ported toward standard OpenMP Tools (OMPT) interfaces; so it can run on any OpenMP runtime and reach a broader audience than MPC-OMP niche users. The porting had been initiated and would only require further engineering work.

Secondly, our analysis passes do not scale with the number of processing units and performances are rather poor. On scalability, computing the time breakdown (work/idle/overhead) is a rather sequential task anyway: each event must be treated sequentially, so the time frame in between can be classified as 'idleness' or 'overhead' depending on the virtual scheduler state. On performances, reading trace files with our analysis tools showed to be particularly slow. Even filtering-out traced events at run-time, converting the bytecode trace files to CPython `PY_Object` represents from 20% to 40% of the total analysis time. Removing memory conversion costs could

be achieved improving the trace file format, or performing *in-situ* analysis (similarly to `libyt`[1]), dedicating cores to performance profiling; but it may increase interference with the execution over the current design (records are enqueued and flush to disk on program termination).

Finally, our profiler framework provides Python scripting interfaces to create new post-mortem analysis passes. Such scripting approach had already been proposed by V. G. Pinto [3] in StarPU, a source of motivation being the flexibility by a scripting language that allows adapting visualizations and analysis. Our scripting framework could serve future research work on mixed-use of MPI and (MPC)-OMP using dependent tasks: for instance, an idea could be extending I. Daoudi work on task-based OpenMP application performance simulation [170] to simulate not only shared-memory but also distributed execution.

Finally, another direction to explore is static graph analysis to detect unoptimized pattern that could slow down the discovery speed, such as multiple edges or edges redundant by transitivity.

Ensuring the Correct Order of Execution As we ported LULESH and HPCG, most of the efforts were on setting up the correct dependencies between tasks. Ensuring the correct order of execution through dependencies (built upon data-flow) is significantly harder than fork-join parallelization (built upon control-flow). Any mistakes would lead to wrong calculations: debugging is painful and curretly mostly done by adding `taskwait` barriers to enforce execution order. This difficulty had already been characterized by S. Royuela et al. [171] in 2015, which lead them to design compile-time analysis of dependencies to provide correctness tips to programmers. However, as a static analysis tool, their approach was limited to what the compiler can see. It lead them to extend this work in [172] (2020) with Dynamic Binary Instrumentation (DBI) to detect any mistakes in dependencies expression at run-time by tracking tasks memory accesses. Standardizing such tool for OpenMP is necessary to reduce debugging efforts of dependent task-based programs. Following S. Royuela approach, additional run-time analysis on the task dependency graph could be conceived. For instance, we have shown in Section 6.2.3.1 that removing unnecessary dependencies (while preserving the correct order of execution) improves the task graph discovery speed and overall performances. It is not always trivial for programmers and mixed-use of static and dynamic analysis could help.

7.2.2 Compiler and Runtime Collaboration on Memory Prefetching

Performance gains on LULESH and HPCG of Section 6.2 were mostly from taking advantage of temporal locality using a depth-first scheduling heuristic. Applications are memory-bound, so accelerating memory access significantly improves performance. In order to accelerate memory accesses further-more, a way could be cache prefetching. For instance, attaching actual memory accesses performed to each task could lead to earlier and more accurate cache prefetching by the runtime; over traditional compiler/hardware prediction techniques.

In our porting of the application, we built task dependencies upon actual memory accesses; hence the OpenMP runtime even already has this information. Hence, it could perform explicit prefetching (`__builtin_prefetch`[2]) likely earlier than a compiler could detect by itself. Potential benefits would be improving the overlapping of data movements in the memory hierarchy with the execution of independent instructions with earlier prefetching. However, the co-existence of hardware/compiler/runtime memory prefetching could interfere and we believe collaboration of each layer may be needed to enhance performances.

7.2.3 Orchestrating Strong and Weak Progress of Asynchronous Operations

Supercomputers are getting massively parallel and heterogeneous: many-cores GPP, GPUs, FPGAs, NICs, someday Quantum Processing Units (QPUs). In order to fully use the hardware

[1] `https://github.com/yt-project/libyt`

[2] `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`

capability, programmers must use multiple programming models jointly, usually:

- one per device type,
- each requiring CPU time for transition system initialization and progression, which can overlap one another

For instance, MPI requires CPU time to "*to respond to control messages, post network read/write operations or poll the Host Channel Adapter (HCA), and orchestrate collective operations.*" [173]; or Cuda requires CPU time to construct and schedule Cuda Graph, or perform memory transfers. Asynchronous operation progression mechanisms are often classified as

- strong - if it bounds the asynchronous operation duration,
- weak - else (if no such bound is guaranteed).

Strong progress is usually implemented through dedicated resources (physical cores, or kernel threads preempting the execution [6, 102]). It can lead to interferences degrading performances of each programming model [111, 174]: dedicating a physical core restricts the number of workers for each programming models, and ovesubscribing with kernel threads can deteriorate performances of each thread sharing the same core [174, 175]. In this manuscript, we have shown the importance of the task dependency graph discovery: sharing CPU time between producer threads and a progression thread would probably severely deteriorate performances of discovery-bound applications. On the other hand, weak progress is usually implemented with opportunistic polling, such as MPC collaborative polling on idle periods [77] or in-between OpenMP task scheduling decisions as we proposed in this thesis. Such approach can limit interferences removing kernel preemption with a 1:1 threads/core binding model; but it is not sufficient to bound asynchronous operation duration. Hence, a research direction to pursue is the orchestration of both strong and weak progress to provide both a duration time bound while limiting interferences at most.

7.2.4 Speculative Execution, and Distributed Task Graph Cancellation

Parallel *iterative* applications consists in re-iterating a computation scheme until a solution converged. The solution convergence test usually requires a global synchronization. Hence, it can be tested whether naively synchronizing on each iteration, or every few iteration as a tradeoff between the synchronization overheads and induced extra-computation [176].

In Section 6.1, we presented our task-based porting of the LULESH and HPCG applications which are both iterative. LULESH and HPCCG criteria are the following:

- LULESH - the simulation finished (on the time-axis).
- HPCG - the residual $r_{k+1} = r_k - a_k.A.p_k$ is small-enough, providing an acceptable solution.

On the reference `parallel-for` versions, stopping the execution is straightforward due to the really-synchronous structure of the code, simply testing criteria after each iteration barrier. However, in the task-based versions, such barriers are removed, and criteria are only known after executing a specific task:

- LULESH - the task computing the time-step for the next iteration; line 4 on Listing 6.7.
- HPCG - the task computing the residual; represented in orange in the middle of Fig. 6.2.

When such "stopping criterion tasks" execute, tasks from many more iterations may already have been discovered by the producer thread. This in particular interesting to overlap (a) the task graph discovery with computation and (b) the completion of the iteration i with independent work

from iterations $j > i$. (b) is also known as *speculative computation*: even though the execution of tasks from iteration $i + 1$ may not be needed, they are still executed speculatively, assuming the next iteration is actually needed. Speculative computation on LULESH is very-visible on the Gantt chart Fig. 6.14, where tasks of the same iteration are grouped per color.

However, once the stopping criterion is met, tasks from future iterations must be somehow canceled. As part of this thesis, we faced two difficulties to cancel a distributed task dependency graph with MPI and OpenMP under the current standard specifications. The first one is related to shared-memory cancellation with OpenMP, and the second one is related to on-going MPI communications cancellation.

Canceling OpenMP Tasks OpenMP provides a task cancellation interface. Using the OpenMP task transition system Fig. 3.2, the semantic of the `cancel` construct with specifications 5.2 is the following; after executing the construct:

- scheduled tasks finish execution; transition ④ → ⑥.
- un-scheduled tasks does not execute; transition ④ → ⑦,⑧.

However, it remains the programmer responsibility to ensure the consistency of the computed solution, even if cancelled in the midsts of an interation.

Canceling MPI Requests Though, not only local tasks but also on-going communications and remote tasks must be cancelled. MPI specifications provide the `MPI_Cancel` interface to cancel a request. However, it only specifies its use on point-to-point receive requests[3]. Canceling point-to-point send requests used to be specified by MPI, but led to implementation issues[4]. Supporting cancellation on send requests implied overhead on every request, and there were no clear motivating use cases, leading the committee to remove it from the specification. In the context of speculative computation with task-based mixed use of MPI+OpenMP, canceling send operations and even collectives could be a motivating use case.

Nevertheless, canceling each request individually still implies unnecessary overheads on historical use cases (that would never cancel), and does not provide solutions to remote OpenMP tasks cancellation. That's why exploring cancellation on a coarser level (communicator or session) is probably a more suitable solution.

Going Forward In this thesis, we always ensured that stopping criteria were never met: in all our evaluations, the application always executes in a fixed amount of iterations, hence avoiding cancellation difficulties. Note that LULESH leap/frog solver (at most 2 different iterations on the fly), and the conjugate gradient (at most 1 iteration on the fly) remains inherently synchronous algorithm due to global reductions, limiting speculative computation possibilities. Asynchronous algorithm could benefit more from such speculative tasking approach. Providing solutions to mentionned difficulties and studying the impacts of distributed cancellation is a future work to be conducted to assess on benefits and drawbacks.

7.2.5 Co-Scheduling Task-based Programs to Maximize Resources Usage

As we discussed in our publication [177], executing multiple programs on the same compute node is a way to maximize compute resources use. The general idea is to overlap the idleness of a given application with useful work from another application. The first possibility is through cores over-subscription with kernel threads (e.g., *pthreads*), letting the operating system preemptively schedule threads. With n cores and $m > n$ threads, this is the $m : n$ co-scheduling approach.

[3] `https://www.mpich.org/static/docs/v3.2/www3/MPI_Cancel.html`
[4] `https://github.com/mpi-forum/mpi-forum-historic/issues/480`

Over-subscription had been widely adopted in parallel runtimes for overlapping synchronization idleness on cores, for instance, in AMPI [178] or LLVM OpenMP [67]. Yet, it can degrade performances when a high number of asynchronous operations concurrently progressed [175, 179], one source of overhead being the operating system.

The second possibility is an $n : n$ approach: each core is assigned a single kernel thread, and the scheduling decision is taken by a user-space scheduler *cooperatively* (such as MPC). A motivating result is Figure 8.2 of C. Barbossa Ph.D. thesis [180]. Her objective was to reduce idleness induced by OpenMP synchronization from unbalanced work between threads of a `parallel for` loop. In her experiment, she compared the performances of three scenarios co-scheduling two applications (MiniFE and Quicksilver), showing that letting the operating system manage scheduling is not sufficient to maximize resource usage in the HPC proxy-applications understudy. Precise reasons remain to be investigated but could include, for instance, preemption costs or CPU resource retention during synchronizations (spinlock).

A research direction we would like to conduct in the future is the co-scheduling of multiple task-based applications.

7.2.6 Task-based MPI and OpenMP Composition: Who is the Audience ?

Throughout this thesis, we have demonstrated that the mixed-use of MPI and OpenMP could respond to the problem of performance portability. We provided a few answers to the programming, profiling, and performance difficulties we introduced; however, hardware complexity remains: heterogeneity, hierarchical memory, interconnection network... As hardware becomes more complex, standard interface follows, and programming both performant and portable applications become almost impossible for a non-expert audience. In order to ease programming to a non-HPC expert audience and fully embrace the problem of performance portability, a promising long-term solution consists in designing higher-level and constrained interfaces such as Domain Specific Languages (DSL) on top of MPI and OpenMP: MPI and OpenMP would target HPC experts conceiving DSLs; while DSLs target scientific simulation programmers. In the literature, a few have initiated this approach, such as NabLab [43] for simulations over unstructured mesh. In the future, designing, optimizing, and maybe even unifying DSL are research directions to conduct for the performance portability of scientific simulations.

Annexes

Chapter 8

Annexes

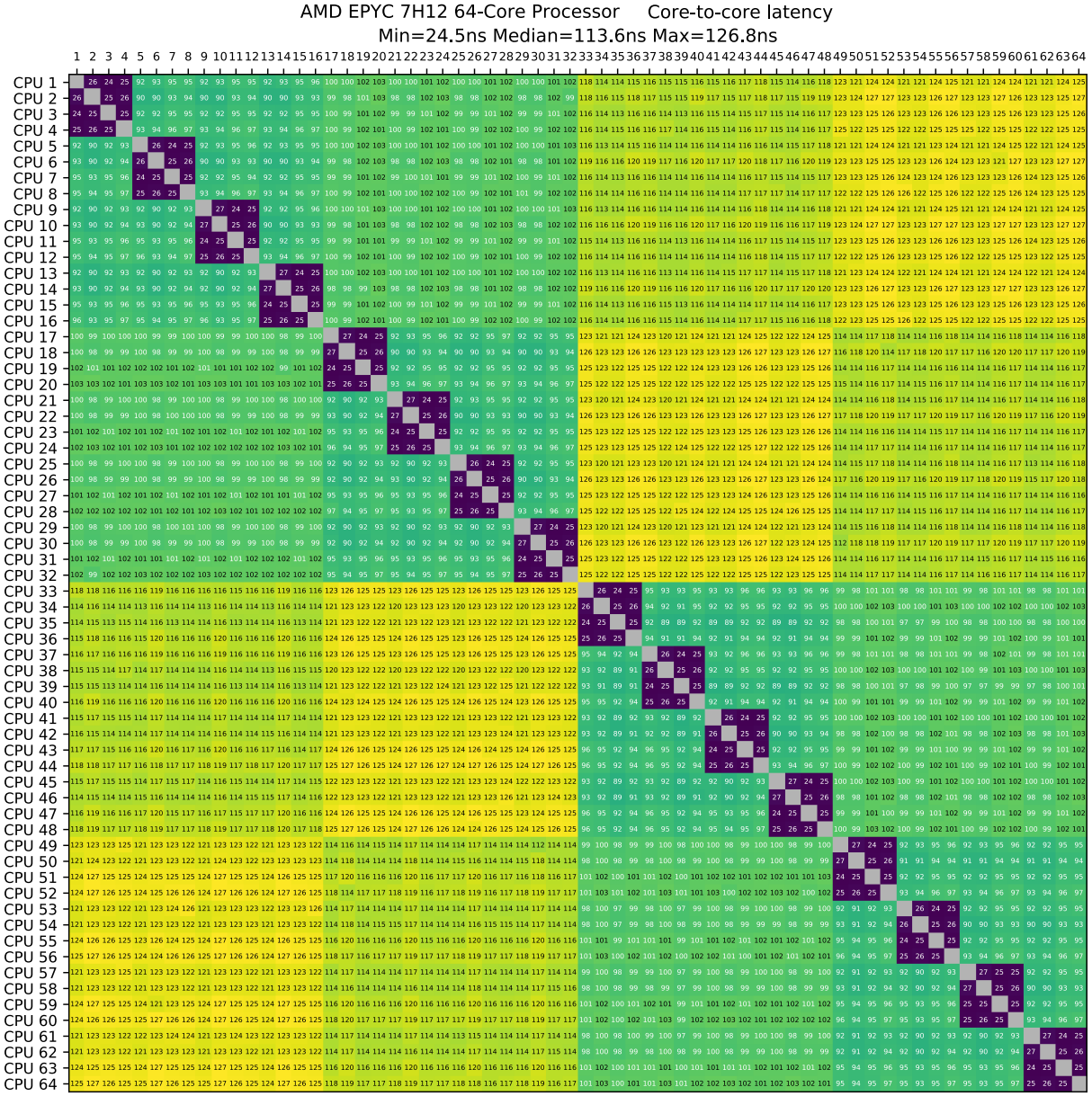


Figure 8.1: Hierarchical Memory for a Processor

```
// User code
# pragma omp task context(stack-size)
{
    [...]
    MPI_Start(&req);
    [...]
    TAMPI_Wait(&req, &status);
    [...]
}

// MPC-OMP Portability Interface
int
progress(progress_info_t * infos)
{
    int completed;
    MPI_Test(infos.req, &flag, infos.statuses);
    if (completed)
        omp_fulfill_event(infos->ev_handle;)
    return completed ? MPC_OMP_CANCEL : MPC_OMP_REQUEUE;
}

int
TAMPI_Wait(MPI_Request * req, MPI_Status * status)
{
    omp_event_handle_t ev_handle = mpc_omp_task_continuation_event();

    progress_info_t infos = {
        .req       = req,
        .status    = status,
        .ev_handle = ev_handle,
    };
    mpc_omp_callback(progress, &infos, OMP_CALLBACK_TASK_SCHEDULE);

    mpc_omp_taskwait_detach(ev_handle);
}
```

Listing 8.1: MPI and OpenMP runtime interoperability approach

```
# include <stdio.h>

int
main(void)
{
    # pragma omp parallel
    {
        # pragma omp single
        {
            int x = 0; (void) x;
            # pragma omp task depend(out:  x)
                while (1);

        while (1)
        {
            # pragma omp task depend(in:  x)
            {}
            if (++x % 100 == 0) printf("submitted␣%d␣tasks\n", x);
        }
    }
    return 0;
}
```

Listing 8.2: A program generating a lot of dependant tasks

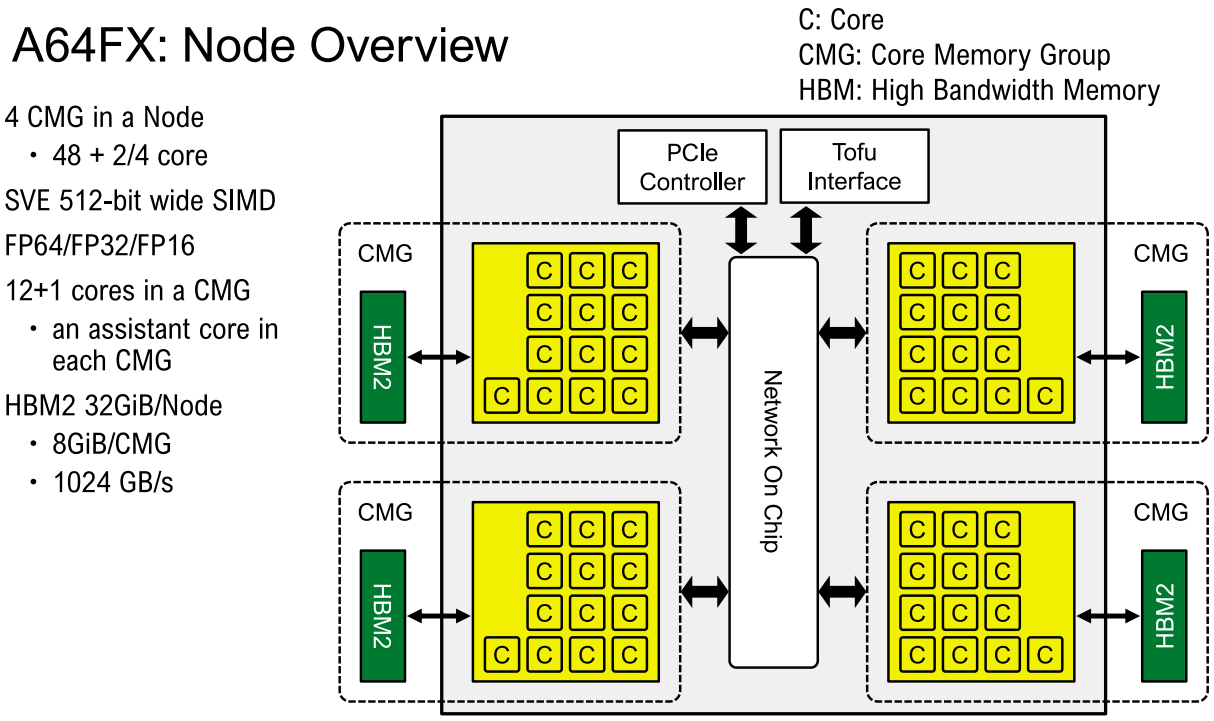


Figure 8.2: Fugaku's A64FX Compute Node

LULESH Detailed task grain

Task Label	Work (µs)	TOT_INS/TOT_CYC	DP_OPS/LST_INS
AABCForNodes	4307	0.03	2E-05
ApplyMaterialPropertiesForElems	68.18	1.97	1.98
CalcAccelerationForNodes	726.41	0.27	0.33
CalcCourantConstraintForElems	639.95	0.36	0.25
CalcCourantConstraintForElems_reduce	3.14	0.42	0.01
CalcEnergyForElems1	595.48	0.25	2.32
CalcEnergyForElems2	1077.53	0.58	1.12
CalcEnergyForElems3	209.08	0.55	1.99
CalcEnergyForElems4	1274.39	0.61	1.12
CalcEnergyForElems5	905.58	0.49	0.73
CalcFBHourglassForceForElems1	18344.26	1.9	1.41
CalcFBHourglassForceForElems2	5478.5	0.23	0.67
CalcForceForNodes	246.13	0	0.01
CalcHourglassControlForElems	22436.32	1.98	0.63
CalcHydroConstraintForElems	210.4	0.72	0
CalcHydroConstraintForElems_reduce	1.18	0.5	0.01
CalcKinematicsForElems	7210.77	2.69	1.08
CalcLagrangeElements	281.53	0.6	1.2
CalcMonotonicQGradientsForElems	4256.46	2.15	0.87
CalcMonotonicQRegionForElems	3089.27	1.15	1
CalcPositionForNodes	440.9	0.57	0.67
CalcPressureForElems1	315.06	0.22	1.32
CalcPressureForElems2	548.12	0.53	0.33
CalcSoundSpeedForElems	590.05	0.42	0.86
CalcTimeConstraintsForElems_init	0.5	0.74	0.01
CalcTimeConstraintsForElems_reduce_courant	1.5	0.23	0.01
CalcTimeConstraintsForElems_reduce_hydro	1	0.36	0.01
CalcVelocityForNodes	483.58	0.94	0.67
EvalEOSForElems1	1988.75	0.15	0
EvalEOSForElems2	676.42	0.46	1
EvalEOSForElems3	249.7	0.39	0
EvalEOSForElems4	124.25	0.92	0
EvalEOSForElems5	80.25	0.01	0.01
EvalEOSForElems6	961.96	0.2	0
InitStressTermsForElems	1028.95	0.22	0.4
IntegrateStressForElems1	14786.62	2.58	0.66
IntegrateStressForElems2	5549.58	0.22	0.65
TimeDump	174.5	0.13	0
TimeIncrement	5.5	0.29	0.02
UpdateVolumesForElems	143.54	1.18	0.5

Table 8.1: LULESH per-task: (1) average work, (2) instructions per cycle, and (2) double operation per load/store instructions ; for -s 384 -tel 1024 -tnl 1024 on 24 Intel(R) Xeon(R) Platinum 8168

```
1  # include <stdlib.h>
2  # include <stdio.h>
3  # include <time.h>
4  # include <omp.h>
5
6  # define N 1000000
7
8  int
9  main(void)
10 {
11     srand(time(NULL));
12
13     // warm-up runtime
14     # pragma omp parallel
15     {
16         int i;
17         for (i = 0 ; i < 16 ; ++i)
18         {
19             int p = rand();
20             # pragma omp task priority(p)
21             {}
22         }
23     }
24
25     // benchmark
26     const double t0 = omp_get_wtime();
27     # pragma omp parallel
28     {
29         int i;
30         for (i = 0 ; i < N ; ++i)
31         {
32             int p = rand();
33             # pragma omp task priority(p)
34             {}
35         }
36     }
37     const double tf = omp_get_wtime();
38     printf("took␣%lf\n", tf - t0);
39
40     return 0;
41 }
```

Listing 8.3: Microbenchmark on the priority clause

```
1  # include <omp.h>
2
3  static volatile int T1_completed  = 0;
4  static volatile int T2_started   = 0;
5
6  static void
7  T1(void)
8  {
9      while (!T2_started)
10     {
11         # pragma omp taskyield
12     }
13     T1_completed = 1;
14 }
15
16 static void
17 T2(void)
18 {
19     T2_started = 1;
20     while (!T1_completed)
21     {
22         # pragma omp taskyield
23     }
24 }
25
26 int
27 main(void)
28 {
29     # pragma omp parallel
30     {
31         // block every threads but thread 0, to force schedule on thread 0
32         while (!T1_completed && omp_get_thread_num() != 0);
33
34         # pragma omp single
35         {
36             // send 'T2' then 'T1', as LLVM follows a LIFO strategy,
37             // so 'T1' is scheduled first
38
39             # pragma omp task untied
40                 T2();
41
42             # pragma omp task untied
43                 T1();
44         }
45     }
46     return 0;
47 }
```

Listing 8.4: Valid OpenMP program deadlocking with LLVM

Bibliography

[1] Tjarda Boekholt, Valerio Ribeiro, Bruno Coelho, Domingos Barbosa, Sonia Anton, and Alexandre Correia. Multi-physics simulations in astronomy. `https://utaustinportugal.org/wp-content/uploads/2019/10/UTAPC053_AdvancedComputing.pdf`, 2019. (cited on page 8)

[2] Takemasa Miyoshi, Masaru Kunii, Juan Ruiz, Guo-Yuan Lien, Shinsuke Satoh, Tomoo Ushio, Kotaro Bessho, Hiromu Seko, Hirofumi Tomita, and Yutaka Ishikawa. "big data assimilation" revolutionizing severe weather prediction. *Bulletin of the American Meteorological Society*, 97(8):1347 – 1354, 2016. (cited on page 8)

[3] Vinicius Garcia Pinto, Lucas Mello Schnorr, Luka Stanisic, Arnaud Legrand, Samuel Thibault, and Vincent Danjean. A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. *Concurrency and Computation: Practice and Experience*, 30(18):1–31, April 2018. (cited on page 10, 44, 104)

[4] Larry Meadows and Ken-ichi Ishikawa. Openmp tasking and mpi in a lattice qcd benchmark. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 77–91, Cham, 2017. Springer International Publishing. (cited on page 10, 36, 52, 53, 69)

[5] Eric Aubanel. Parallel program comprehension. In Mariana Marasoiu, Colin B. D. Clark, Philip Tchernavskij, Ben Shapiro, Clayton Lewis, and Luke Church, editors, *Proceedings of the 31st Annual Workshop of the Psychology of Programming Interest Group, PPIG 2020, Online, Summer Edition: August 17-21, Winter Edition: December 1-4, 2020*. Psychology of Programming Interest Group, 2020. (cited on page 10)

[6] 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*, 85:153–166, 2019. (cited on page 10, 35, 37, 38, 40, 53, 105)

[7] Thierry Gautier, Christian Pérez, and Jérôme Richard. *On the Impact of OpenMP Task Granularity: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings*, pages 205–221. 01 2018. (cited on page 10, 76, 99)

[8] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 235–244, New York, NY, USA, 2004. Association for Computing Machinery. (cited on page 10, 43, 86)

[9] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012. (cited on page 10, 42, 43, 65)

[10] Oliver Sinnen and Leonel Sousa. Communication contention in task scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 16:503– 515, 07 2005. (cited on page 10, 42)

[11] Marc Perache, Herve Jourdren, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th International Euro-par Conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg, 2008. Spinger-Verlag. (cited on page 10, 30, 39, 54, 67)

[12] Ricardo Fonseca, S. Martins, L. Silva, J. Tonge, Frank Tsung, and W. Mori. One-to-one direct modeling of experiments and astrophysical scenarios: Pushing the envelope on kinetic plasma simulations. *Plasma Physics and Controlled Fusion*, 50, 10 2008. (cited on page 13)

[13] SCM - HPC center. `https://www.scm.com/applications/a-computing-center/`, 2023. [Online; accessed 28-June-2023]. (cited on page 13)

[14] ETP4HPC - What is HPC ? `https://www.etp4hpc.eu/what-is-hpc.html`, 2023. [Online; accessed 28-June-2023]. (cited on page 13)

[15] Patrick Carribault. *Compiler/Runtime Cooperation for High-Performance Multi-Paradigm Parallelism - Habilitation à Diriger des Recherches*. 2015. (cited on page 14)

[16] Julian Shun. 6.172 performance engineering of software systems - caching and cache-efficient algorithms, 2018. (cited on page 15)

[17] Stefan G. Berg. Cache prefetching (technical report uw-cse 02-02-04). Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350, 2002. (cited on page 15)

[18] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, jun 2000. (cited on page 15)

[19] David A Patterson, John L Hennessy, and David Goldberg. *Computer architecture : a quantitative approach*. 1996. (cited on page 15)

[20] Tim Woodall, Galen Shipman, George Bosilca, Richard Graham, and Arthur Maccabe. High performance rdma protocols in hpc. pages 76–85, 09 2006. (cited on page 16)

[21] Justin L. Whitt. Frontier overview - olcf annual user meeting, October 2022. (cited on page 16)

[22] Manuel Ferat, Romain Pereira, Adrien Roussel, Patrick Carribault, Luiz-Angelo Steffenel, and Thierry Gautier. Enhancing mpi+openmp task based applications for heterogeneous architectures with gpu support. In Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth, editors, *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, pages 3–16, Cham, 2022. Springer International Publishing. (cited on page 16, 55, 89, 103)

[23] The Open Group. Posix base specifications issue 7, 2018 edition ieee std 1003.1-2017. (cited on page 16)

[24] Robert Love. *Linux Kernel Development: Linux Kernel Development _p3*. Pearson Education, 2010. (cited on page 17, 18)

[25] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014. (cited on page 17)

[26] Greg Regnier, Srihari Makineni, I. Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and A. Foong. Tcp onloading for data center servers. *Computer*, 37:48 – 58, 12 2004. (cited on page 17)

[27] Saïd Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. The bxi interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, 2015. (cited on page 18)

[28] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association. (cited on page 18)

[29] John Backus. The history of fortran i, ii and iii. In *History of Programming Languages*, pages 25–74. Association for Computing Machinery, 1978. (cited on page 19)

[30] Dennis Ritchie and Brian Kernighan. *The C Programming Language*, 1978. (cited on page 20)

[31] Bjarne Stroustrup. The c++ programming language - reference manual. In *Computing Science Technical Report No 108*, 1984. (cited on page 20)

[32] Michel F. Sanner. Python: a programming language for software integration and development. *Journal of molecular graphics & modelling*, 17 1:57–61, 1999. (cited on page 20)

[33] Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W. J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. An extreme-scale implicit solver for complex pdes: Highly heterogeneous flow in earth's mantle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. (cited on page 21)

[34] Wayne Joubert, Deborah Weighill, David Kainer, Sharlee Climer, Amy Justice, Kjiersten Fagnan, and Daniel Jacobson. Attacking the opioid epidemic: Determining the epistatic and pleiotropic genetic architectures for chronic pain and opioid addiction. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 717–730, 2018. (cited on page 21)

[35] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, Weinan E, and Linfeng Zhang. Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. (cited on page 21)

[36] Luca Fedeli, Axel Huebl, France Boillod-Cerneux, Thomas Clark, Kevin Gott, Conrad Hillairet, Stephan Jaure, Adrien Leblanc, Rémi Lehe, Andrew Myers, Christelle Piechurski, Mitsuhisa Sato, Neil Zaïm, Weiqun Zhang, Jean-Luc Vay, and Henri Vincenti. Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. (cited on page 21)

[37] Dennis M. Ritchie. *The Development of the C Programming Language*, page 671–698. Association for Computing Machinery, New York, NY, USA, 1996. (cited on page 21)

[38] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA, 2009. (cited on page 21)

[39] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008. (cited on page 21)

[40] D.B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(1):25–42, 1993. (cited on page 22)

[41] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 7–1–7–22, New York, NY, USA, 2007. Association for Computing Machinery. (cited on page 22)

[42] Valentin Churavy, William Godoy, Carsten Bauer, Hendrik Ranocha, Michael Schlottke-Lakemper, Ludovic Räss, Johannes Blaschke, Mosè Giordano, Erik Schnetter, Samuel Omlin, Jeffrey Vetter, and Alan Edelman. Bridging hpc communities through the julia programming language, 11 2022. (cited on page 22)

[43] Benoit Lelandais, Marie-Pierre Oudot, and Benoit Combemale. Fostering metamodels and grammars within a dedicated environment for hpc: The nablab environment (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, page 200–204, New York, NY, USA, 2018. Association for Computing Machinery. (cited on page 22, 66, 107)

[44] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3):1165–1187, 2019. (cited on page 22)

[45] Sven Karol, Tobias Nett, Jeronimo Castrillon, and Ivo F. Sbalzarini. A domain-specific language and editor for parallel particle methods. *ACM Trans. Math. Softw.*, 44(3), mar 2018. (cited on page 22)

[46] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. (cited on page 22, 23, 24, 28, 31)

[47] Jonathan A. Thompson and Kristofer Schlachter. An introduction to the opencl programming model. 2012. (cited on page 22)

[48] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021. (cited on page 22, 29)

[49] David Beckingsale, Thomas Scogland, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam Kunen, Olga Pearce, Peter Robinson, and Brian Ryujin. Raja: Portable performance for large-scale scientific applications. pages 71–81, 11 2019. (cited on page 23, 66)

[50] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022. (cited on page 23, 66)

[51] CEA/IFPEN and Contributors. Arcane. `https://github.com/arcaneframework/framework`, 2023. (cited on page 23)

[52] Thomas Willhalm and Nicolae Popovici. Putting intel® threading building blocks to work. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, page 3–4, New York, NY, USA, 2008. Association for Computing Machinery. (cited on page 23, 24)

[53] G. Bosilca, R.J. Harrison, T. Herault, M.M. Javanmard, P. Nookala, and E.F. Valeev. The template task graph (ttg) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale. In *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 1–7, 2020. (cited on page 23, 24)

[54] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011. (cited on page 23, 24, 51, 99)

[55] Michael Bauer. Legion: programming distributed heterogeneous architectures with logical regions. 2014. (cited on page 23, 24)

[56] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx – a task based programming model in a global address space. 10 2014. (cited on page 23, 24)

[57] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, 15(2):66–71, 2013. (cited on page 23, 24)

[58] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. (cited on page 23, 24)

[59] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. New York, NY, USA, 2005. Association for Computing Machinery. (cited on page 23, 24)

[60] Thierry Gautier, Joao V.F Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308, 2013. (cited on page 23, 25, 31, 90, 99)

[61] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, page 88, USA, 1998. IEEE Computer Society. (cited on page 23, 25, 90, 99)

[62] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-taskflow: Fast task-based parallel programming using modern c++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 974–983, 2019. (cited on page 23, 25)

[63] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A

taxonomy of task-based parallel programming technologies for high-performance computing. *J. Supercomput.*, 74(4):1422–1434, apr 2018. (cited on page 23, 24, 25, 26)

[64] Robert Keller. A fundamental theorem of asynchronous parallel computation. *LNCS*, 24:102–112, 01 1975. (cited on page 23)

[65] David Álvarez and Vicenç Beltran. Optimizing iterative data-flow scientific applications using directed cyclic graphs. *IEEE Access*, 11:51971–51984, 2023. (cited on page 24)

[66] Atmn Patel and Johannes Doerfert. Remote openmp offloading. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 441–442, New York, NY, USA, 2022. Association for Computing Machinery. (cited on page 25)

[67] Shilei Tian, Johannes Doerfert, and Barbara Chapman. Concurrent execution of deferred openmp target tasks with hidden helper threads. In Barbara Chapman and José Moreira, editors, *Languages and Compilers for Parallel Computing*, pages 41–56, Cham, 2022. Springer International Publishing. (cited on page 25, 107)

[68] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical report, USA, 1992. (cited on page 25)

[69] Thierry GAUTIER. Contribution à la définition des supports logiciels pour une programmation et une exécution portable et efficace en hpc. 2019. (cited on page 28)

[70] Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9, 08 2000. (cited on page 28, 35)

[71] Ryan Grant, Matthew Dosanjh, Michael Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded mpi communication. pages 330–350, 05 2019. (cited on page 29, 52)

[72] Hugo Taboada. *Recouvrement des Collectives MPI Non-bloquantes sur Processeur Manycore*. PhD thesis, 2018. Thèse de doctorat dirigée par Jeannot, Emmanuel et Denis, Alexandre Informatique Bordeaux 2018. (cited on page 30)

[73] Florian Reynier. *A study on progression of MPI communications using dedicated resources*. PhD thesis, 2022. Thèse de doctorat dirigée par Jeannot, Emmanuel et Denis, Alexandre Informatique Bordeaux 2022. (cited on page 30)

[74] Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache, and Hugo Taboada. Study on progress threads placement and dedicated cores for overlapping mpi nonblocking collectives on manycore processor. *The International Journal of High Performance Computing Applications*, 33:109434201986018, 07 2019. (cited on page 30)

[75] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. Bluesmpi: Efficient mpi non-blocking alltoall offloading designs on modern bluefield smart nics. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*, page 18–37, Berlin, Heidelberg, 2021. Springer-Verlag. (cited on page 30)

[76] Alexandre Denis. pioman: A pthread-based multithreaded communication engine. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 155–162, 2015. (cited on page 30)

[77] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. Improving mpi communication overlap with collaborative polling. volume 96, 09 2012. (cited on page 30, 39, 105)

[78] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian W. Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *PVM/MPI*, 2004. (cited on page 30)

[79] Ewing Lusk. User's guide for mpich, a portable implementation of mpi. 06 2000. (cited on page 30)

[80] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. New madeleine: a fast communication scheduling engine for high performance networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007. (cited on page 30)

[81] Derek Schafer, Ryan Marshall, Tony Skjellum, and Martin Ruefenacht. Overview of ExaMPI. (cited on page 30)

[82] LLVM/OpenMP Documentation. `https://openmp.llvm.org/`, 2023. [Online; accessed 30-March-2023]. (cited on page 31)

[83] GCC Wiki - OpenMP. `https://gcc.gnu.org/wiki/openmp`, 2023. [Online; accessed 30-March-2023]. (cited on page 31)

[84] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In Mitsuhisa Sato, Toshihiro Hanawa, MatthiasS. Müller, BarbaraM. Chapman, and BronisR. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*, volume 6132 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2010. (cited on page 31, 32, 39)

[85] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018. (cited on page 31, 67)

[86] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Task-based programming with ompss and its application. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 601–612, Cham, 2014. Springer International Publishing. (cited on page 31, 90, 99)

[87] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 212–223, New York, NY, USA, 1998. Association for Computing Machinery. (cited on page 31, 48, 99)

[88] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009. (cited on page 31)

[89] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in OpenMP. In Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, pages 1–12, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. (cited on page 31)

[90] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. Extending the openmp tasking model to allow dependent tasks. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, pages 111–122, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. (cited on page 31, 99)

[91] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, pages 16–29, Cham, 2014. Springer International Publishing. (cited on page 31, 71)

[92] J Balart, A Duran, Mg Gon, Xavier Martorell, E Ayguadé, and Jesús Labarta. Nanos mercurium: A research compiler for openmp. *Proceedings of the European Workshop on OpenMP*, 8, 01 2004. (cited on page 32)

[93] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008. (cited on page 34, 57, 58)

[94] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A proposal to extend the openmp tasking model for heterogeneous architectures. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, pages 154–167, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (cited on page 34)

[95] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, aug 1990. (cited on page 35)

[96] Miwako Tsuji and Mitsuhisa Sato. Performance evaluation of openmp and mpi hybrid programs on a large scale multi-core multi-socket cluster, t2k open supercomputer. In *2009 International Conference on Parallel Processing Workshops*, pages 206–213, 2009. (cited on page 35)

[97] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 5–16, New York, NY, USA, 2010. Association for Computing Machinery. (cited on page 36, 44, 52)

[98] Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Effective communication and computation overlap with hybrid mpi/smpss. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP

'10, page 337–338, New York, NY, USA, 2010. Association for Computing Machinery. (cited on page 36)

[99] Joseph Schuchart, Keisuke Tsugane, José Gracia, and Mitsuhisa Sato. The impact of taskyield on the design of tasks communicating through mpi. In Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 3–17, Cham, 2018. Springer International Publishing. (cited on page 36, 49, 52, 56, 69, 98)

[100] Huiwei Lu, Sangmin Seo, and Pavan Balaji. Mpi+ult: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454, 2015. (cited on page 37)

[101] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. Bolt: Optimizing openmp parallel regions with user-level threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 29–42, 2019. (cited on page 37, 55, 67)

[102] Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S. Müller, and Torsten Kuhlen. Mpi detach - asynchronous local completion. In *Proceedings of the 27th European MPI Users' Group Meeting*, EuroMPI/USA '20, page 71–80, New York, NY, USA, 2020. Association for Computing Machinery. (cited on page 38, 105)

[103] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. Callback-based completion notification using mpi continuations. *Parallel Computing*, 106:102793, 2021. (cited on page 38)

[104] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Thread-local storage extension to support thread-based mpi/openmp applications. In BarbaraM. Chapman, WilliamD. Gropp, Kalyan Kumaran, and MatthiasS. Müller, editors, *OpenMP in the Petascale Era, Proceedings of the 7th International Workshop on OpenMP (IWOMP 2011)*, volume 6665 of *Lecture Notes in Computer Science*, pages 80–93. Springer Berlin Heidelberg, 2011. (cited on page 39)

[105] Aurèle Mahéo, Souad Koliaï, Patrick Carribault, Marc Pérache, and William Jalby. Adaptive openmp for large numa nodes. In Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, pages 254–257, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (cited on page 39)

[106] Aurèle Maheo. *Improving the Hybrid model MPI+Threads through Applications, Runtimes and Performance tools*. Theses, Université de Versailles-Saint Quentin en Yveslines, Sep 2015. (cited on page 40)

[107] Yves Robert. *Task Graph Scheduling*, pages 2013–2025. Springer US, Boston, MA, 2011. (cited on page 42)

[108] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13:260–274, 04 2002. (cited on page 42)

[109] Unit Mixte, Recherche Lyon, Olivier Beaumont, A. Legr, and Yves Robert. Static scheduling strategies for heterogeneous systems. 05 2003. (cited on page 42)

[110] M.K. Qureshi, D.N. Lynch, O. Mutlu, and Y.N. Patt. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178, 2006. (cited on page 42)

[111] Alexandre Denis, Emmanuel Jeannot, and Philippe Swartvagher. Interferences between communications and computations in distributed hpc systems. In *50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA, 2021. Association for Computing Machinery. (cited on page 42, 44, 105)

[112] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *SIGOPS Oper. Syst. Rev.*, 40(4):403–414, apr 2006. (cited on page 42)

[113] Nathan R. Tallent and John M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. *SIGPLAN Not.*, 44(4):229–240, feb 2009. (cited on page 43, 50)

[114] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. A proposal to extend the openmp tasking model with dependent tasks. *International Journal of Parallel Programming*, 37:292–305, 06 2009. (cited on page 43)

[115] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moreto, Martin Schulz, Ramon Beivide, Mateo Valero, and Abhinav Bhatele. Optimizing computation-communication overlap in asynchronous task-based programs: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 415–416, New York, NY, USA, 2019. Association for Computing Machinery. (cited on page 44)

[116] Jérôme Richard, Guillaume Latu, Julien Bigot, and Thierry Gautier. Fine-grained mpi+openmp plasma simulations: Communication overlap with dependent tasks. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 419–433, Cham, 2019. Springer International Publishing. (cited on page 45, 57, 69)

[117] Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, and Florian Reynier. A methodology for assessing computation/communication overlap of mpi nonblocking collectives. *Concurrency and Computation: Practice and Experience*, 34(22):e7168, 2022. (cited on page 45)

[118] Jonathon Anderson, Yumeng Liu, and John Mellor-Crummey. Preparing for performance analysis at exascale. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. (cited on page 47, 51)

[119] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, may 2006. (cited on page 47, 51)

[120] Te Phhh, Shirley Moore, Jack Dongarra, N. Garner, K. London, and Phil Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14, 07 2000. (cited on page 47)

[121] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. (cited on page 48)

[122] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 362–371, New York, NY, USA, 1993. Association for Computing Machinery. (cited on page 50)

[123] Anton Daumen, Patrick Carribault, François Trahay, and Gaël Thomas. Scalomp: Analyzing the scalability of openmp applications. In Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman, editors, *OpenMP: Conquering the Full Hardware Spectrum*, pages 36–49, Cham, 2019. Springer International Publishing. (cited on page 50)

[124] Idriss Daoudi, Philippe Virouleau, Thierry Gautier, Samuel Thibault, and Olivier Aumage. somp: Simulating openmp task-based applications with numa effects. pages 197–211, 09 2020. (cited on page 50)

[125] Vitoria Pinho, Hervé Yviquel, Marcio Machado Pereira, and Guido Araujo. Omptracing: Easy profiling of openmp programs. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 249–256, 2020. (cited on page 50)

[126] Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. Mercurium: Design decisions for a s2s compiler. 2011. (cited on page 51)

[127] J. Chassin de Kergommeaux, B. Stein, and P.E. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, 2000. (cited on page 51)

[128] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 37–48, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (cited on page 51)

[129] Lucas Leandro Nesi, Vinicius Garcia Pinto, Marcelo Cogo Miletto, and Lucas Mello Schnorr. StarVZ: Performance Analysis of Task-Based Parallel Applications. working paper or preprint, October 2020. (cited on page 51)

[130] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In *IWOMP 2021 - 17th International Workshop on OpenMP*, OpenMP : Enabling Massive Node-Level Parallelism (IWOMP 2021), pages 1–15, Bristol, United Kingdom, September 2021. (cited on page 52, 63, 103)

[131] Romain Pereira, Mael Martin, Adrien Roussel, Patrick Carribault, and Thierry Gautier. Suspending OpenMP Tasks on Asynchronous Events: Extending the Taskwait Construc. In *IWOMP 2023 - 19th International Workshop on OpenMP*, OpenMP: Advanced Task-Based, Device and Compiler Programming (IWOMP 2023), Bristol, United Kingdom, September 2023. (cited on page 52, 54, 103)

[132] OpenMP Architecture Review Board. Openmp application programming interface 5.0. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`. (cited on page 53)

[133] Joachim Protze, Marc-André Hermanns, Matthias S Müller, Van Man Nguyen, Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. MPI detach - Towards automatic asynchronous local completion. *Parallel Computing*, 109:102859, March 2022. (cited on page 53)

[134] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, page 4–es, New York, NY, USA, 2007. Association for Computing Machinery. (cited on page 54)

[135] A. Bastoni, B. Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *OSPERT*, pages 33–44, 01 2010. (cited on page 54)

[136] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 270–281, New York, NY, USA, 2014. Association for Computing Machinery. (cited on page 55)

[137] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022. (cited on page 59)

[138] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on numa architectures. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, pages 531–544, Cham, 2016. Springer International Publishing. (cited on page 59, 67)

[139] Hitoshi Murai, Masahiro Nakao, and Mitsuhisa Sato. XcalableMP programming model and language. In Mitsuhisa Sato, editor, *XcalableMP PGAS Programming Language: From Programming Model to Applications*, pages 1–71. Springer Singapore. (cited on page 66)

[140] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.*, 46(1), apr 2020. (cited on page 66)

[141] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. (cited on page 66)

[142] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, 2012. (cited on page 67)

[143] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):30, October 2014. (cited on page 67)

[144] Mike L. Powell, S. R. Kleiman, S. Barton, D. Shan, D. Stein, and M. Weeks. *SunOS Multi-Thread Architecture*, pages 339–372. Springer New York, New York, NY, 1991. (cited on page 67)

[145] Raymond NAMYST. Contribution à la conception de supports exécutifs multithreads performants. 2001. (cited on page 67)

[146] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. Evaluation of openmp task scheduling algorithms for large numa architectures. volume 8632, pages 596–607, 08 2014. (cited on page 67)

[147] Francesco Massimo, Mathieu Lobet, Julien Derouillat, Arnaud Beck, Guillaume Bouchard, Mickael Grech, Fréderic Pérez, and Tommaso Vinci. A task programming implementation for the particle in cell code smilei. In *Proceedings of the Platform for Advanced Scientific*

Computing Conference, PASC '22, New York, NY, USA, 2022. Association for Computing Machinery. (cited on page 69)

[148] Jack Dongarra, Michael Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications*, 30, 08 2015. (cited on page 70, 97)

[149] Ian Karlin. Lulesh programming model and performance ports overview. 2012. (cited on page 70, 77, 78)

[150] Marcos Maroñas, Xavier Teruel, and Vicenç Beltran. Openmp taskloop dependences. In Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg, editors, *OpenMP: Enabling Massive Node-Level Parallelism*, pages 50–64, Cham, 2021. Springer International Publishing. (cited on page 70)

[151] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95, 12 2013. (cited on page 70)

[152] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *The International Journal of High Performance Computing Applications*, 23(3):284–299, 2009. (cited on page 70)

[153] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. In *HeteroPar 2022*, page 12, Glasgow, United Kingdom, August 2022. (cited on page 70)

[154] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15:803–820, 08 2003. (cited on page 70)

[155] Thierry Gautier, Christian Perez, and Jérôme Richard. On the impact of openmp task granularity. In Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 205–221, Cham, 2018. Springer International Publishing. (cited on page 71, 75)

[156] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 661–672, 2018. (cited on page 77)

[157] Charles Noble, Andrew Anderson, Nathan Barton, Jamie Bramwell, Arlie Capps, Michael Chang, Jin Chou, David Dawson, Emily Diana, Timothy Dunn, Douglas Faux, Aaron Fisher, Patrick Greene, Ines Heinz, Yuliya Kanarska, Saad Khairallah, Benjamin Liu, Jon Margraf, Albert Nichols, and Jeremy White. Ale3d: An arbitrary lagrangian-eulerian multi-physics code, 05 2017. (cited on page 77)

[158] Ian Karlin, Jeff Keasler, and Neely Rob. Lulesh 2.0 updates and changes (technical report). 2013. (cited on page 78)

[159] I Karlin, J McGraw, J Keasler, and B Still. Tuning the LULESH Mini-app for Current and Future Hardware. 2013. (cited on page 78)

[160] Troy D Hanson and Arthur O'Dwyer. uthash: A hash table for c structures, 2013. (cited on page 83)

[161] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, 7(1):5–19, mar 1975. (cited on page 83)

[162] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016. (cited on page 86)

[163] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichlerk, Patrick McCormick, and Alex Aiken. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. (cited on page 93, 94)

[164] Chenle Yu, Sara Royuela, and Eduardo Quiñones. *Enhancing OpenMP Tasking Model: Performance and Portability*, pages 35–49. 09 2021. (cited on page 99)

[165] Chenle Yu, Sara Royuela, and Eduardo Quiñones. Taskgraph: A low contention openmp tasking framework, 2022. (cited on page 99)

[166] David Alvarez and Vicenç Beltran. Optimizing iterative data-flow scientific applications using directed cyclic graphs. 2023. (cited on page 99, 100)

[167] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1151–1158, 2011. (cited on page 99)

[168] Michel Cosnard and Emmanuel Jeannot. Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling. *Parallel Processing Letters*, 11(1):151–168, 2001. (cited on page 99)

[169] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances. In *52nd International Conference on Parallel Processing (ICPP 2023)*, volume 52nd International Conference on Parallel Processing (ICPP 2023), Salt Lake City, United States, August 2023. (cited on page 103)

[170] Idriss Daoudi. *Modélisation de performance et simulation d'applications OpenMP*. Theses, Université de Bordeaux, September 2021. (cited on page 104)

[171] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. Compiler analysis for openmp tasks correctness. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery. (cited on page 104)

[172] Simone Economo, Sara Royuela, Eduard Ayguadé, and Vicenç Beltran. A toolchain to verify the parallelization of ompss-2 applications. In Maciej Malawski and Krzysztof Rzadca, editors, *Euro-Par 2020: Parallel Processing*, pages 18–33, Cham, 2020. Springer International Publishing. (cited on page 104)

[173] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha, and Dhabaleswar K. Panda. Efficient asynchronous communication progress for mpi without dedicated resources. In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI'18, New York, NY, USA, 2018. Association for Computing Machinery. (cited on page 105)

[174] Shu-Mei Tseng, Bogdan Nicolae, Franck Cappello, and Aparna Chandramowlishwaran. Demystifying asynchronous i/o interference in hpc applications. *The International Journal of High Performance Computing Applications*, 35(4):391–412, 2021. (cited on page 105)

[175] Petar Radojkovic, Vladimir Cakarevic, Javier Verdú, Alejandro Pajuelo, Roberto Gioiosa, Francisco Cazorla, Mario Nemirovsky, and Mateo Valero. Measuring operating system overhead on cmt processors. pages 133–140, 10 2008. (cited on page 105, 107)

[176] Olivier Beaumont, El Daoudi, Nicolas Maillard, Pierre Manneback, and Jean-Louis Roch. Tradeoff to minimize extra-computations and stopping criterion tests for parallel iterative schemes. 10 2004. (cited on page 105)

[177] Hugo Taboada, Romain Pereira, Julien Jaeger, and Jean-Baptiste Besnard. Towards achieving transparent malleability thanks to mpi process virtualization. In *2nd International Workshop on Malleability Techniques Applications in High-Performance Computing*, 2023. (cited on page 106)

[178] Yu Pei, George Bosilca, Ichitaro Yamazaki, Akihiro Ida, and Jack Dongarra. Evaluation of programming models to address load imbalance on distributed multi-core cpus: A case study with block low-rank factorization. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pages 25–36, 2019. (cited on page 107)

[179] J. Bierbaum, M. Planeta, and H. Hartig. Towards efficient oversubscription: On the cost and benefit of event-based communication in mpi. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–10, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society. (cited on page 107)

[180] Cassandra Rocha Barbosa, Pierre Lemarinier, Guillaume Papauré, Marc Pérache, and Michaël Krajecki. Sabo: Dynamic mpi+openmp resource balancer. In *2022 IEEE/ACM Fifth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 1–8, 2022. (cited on page 107)