Understanding and Guiding the Computing Resource Management in a Runtime Stacking Context

Arthur Loussert

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Arthur Loussert**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Understanding and Guiding the Computing Resource Management in a Runtime Stacking Context

---

Rapportée par :
Allen D. Malony, *Professor*, University of Oregon
Jean-François Méhaut, *Professeur*, Université Grenoble Alpes

Date de soutenance : 18 Décembre 2019

Devant la commission d'examen composée de :
Raymond Namyst, *Professeur*, Université de Bordeaux – Directeur de thèse
Marc Pérache, *Ingénieur-Chercheur*, CEA – Co-directeur de thèse
Emmanuel Jeannot, *Directeur de recherche*, Inria Bordeaux Sud-Ouest – Président du jury
Edgar Leon, *Computer Scientist*, Lawrence Livermore National Laboratory – Examinateur
Patrick Carribault, *Ingénieur-Chercheur*, CEA – Examinateur
Julien Jaeger, *Ingénieur-Chercheur*, CEA – Invité

– 2019 –

Keywords High-Performance Computing, Parallel Programming, MPI, OpenMP, Runtime Mixing, Runtime Stacking, Resource Allocation, Resource Management

Abstract With the advent of multicore and manycore processors as building blocks of HPC supercomputers, many applications shift from relying solely on a distributed programming model (e.g., MPI) to mixing distributed and shared-memory models (e.g., MPI+OpenMP). This leads to a better exploitation of shared-memory communications and reduces the overall memory footprint. However, this evolution has a large impact on the software stack as applications' developers do typically mix several programming models to scale over a large number of multicore nodes while coping with their hiearchical depth. One side effect of this programming approach is runtime stacking: mixing multiple models involve various runtime libraries to be alive at the same time. Dealing with different runtime systems may lead to a large number of execution flows that may not efficiently exploit the underlying resources.

We first present a study of runtime stacking. It introduces stacking configurations and categories to describe how stacking can appear in applications. We explore runtime-stacking configurations (spatial and temporal) focusing on thread/process placement on hardware resources from different runtime libraries. We build this taxonomy based on the analysis of state-of-the-art runtime stacking and programming models.

We then propose algorithms to detect the misuse of compute resources when running a hybrid parallel application. We have implemented these algorithms inside a dynamic tool, called the Overseer. This tool monitors applications, and outputs resource usage to the user with respect to the application timeline, focusing on overloading and underloading of compute resources.

Finally, we propose a second external tool called Overmind, that monitors the thread/process management and (re)maps them to the underlying cores taking into account the hardware topology and the application behavior. By capturing a global view of resource usage the Overmind adapts the process/thread placement, and aims at taking the best decision to enhance the use of each compute node inside a supercomputer. We demonstrate the relevance of our approach and show that our low-overhead implementation is able to achieve good performance even when running with configurations that would have ended up with bad resource usage.

Host Laboratory:
CEA, DAM, DIF, F-91297 Arpajon, France
Inria Bordeaux Sud-Ouest
LaBRI, Université de Bordeaux

Titre Comprendre et Guider la Gestion des Ressources de Calcul dans un Contexte Multi-Modèles de Programmation

Mots-clés Calcul Haute Performance, Programmation Parallèle, MPI, OpenMP, Mélange de Modèles de Programmation, Allocation des Ressources, Gestion des Ressources

Résumé La simulation numérique reproduit les comportements physiques que l'on peut observer dans la nature. Elle est utilisée pour modéliser des phénomènes complexes, impossible à prédire ou répliquer. Pour résoudre ces problèmes dans un temps raisonnable, nous avons recours au calcul haute performance (High Performance Computing ou HPC en anglais). Le HPC regroupe l'ensemble des techniques utilisées pour concevoir et utiliser les supercalculateurs. Ces énormes machines ont pour objectifs de calculer toujours plus vite, plus précisément et plus efficacement.
Pour atteindre ces objectifs, les machines sont de plus en plus complexes. La tendance actuelle est d'augmenter le nombre cœurs de calculs sur les processeurs, mais aussi d'augmenter le nombre de processeurs dans les machines. Les machines deviennent de plus en hétérogènes, avec de nombreux éléments différents à utiliser en même temps pour extraire le maximum de performances. Pour pallier ces difficultés, les développeurs utilisent des modèles de programmation, dont le but est de simplifier l'utilisation de toutes ces ressources. Certains modèles, dits à mémoire distribuée (comme MPI), permettent d'abstraire l'envoi de messages entre les différents nœuds de calculs, d'autres dits à mémoire partagée, permettent de simplifier et d'optimiser l'utilisation de la mémoire partagée au sein des cœurs de calcul.
Cependant, ces évolutions et cette complexification des supercalculateurs à un large impact sur la pile logicielle. Il est désormais nécessaire d'utiliser plusieurs modèles de programmation en même temps dans les applications. Ceci affecte non seulement le développement des codes de simulations, car les développeurs doivent manipuler plusieurs modèles en même temps, mais aussi les exécutions des simulations. Un effet de bord de cette approche de la programmation est l'empilement de modèles ('Runtime Stacking') : mélanger plusieurs modèles implique que plusieurs bibliothèques fonctionnent en même temps. Gérer plusieurs bibliothèques peut mener à un grand nombre de fils d'exécution utilisant les ressources sous-jacentes de manière non optimale.

L'objectif de cette thèse est d'étudier l'empilement des modèles de programmation et d'optimiser l'utilisation de ressources de calculs par ces modèles au cours de l'exécution des simulations numériques. Nous avons dans un premier temps caractérisé les différentes manières de créer des codes de calcul mélangeant plusieurs modèles. Nous avons également étudié les différentes interactions que peuvent avoir ces modèles entre eux lors de l'exécution des simulations.

De ces observations nous avons conçu des algorithmes permettant de détecter des utilisations de ressources non optimales. Enfin, nous avons développé un outil permettant de diriger automatiquement l'utilisation des ressources par les différents modèles de programmation.

Pour illustrer les différentes techniques permettant de mélanger plusieurs modèles de programmation dans un code de calcul, ainsi que les interactions possibles, nous avons introduit des configurations et catégories de mélange. Cette taxonomie est la base des travaux de cette thèse ainsi que sa première contribution. Les catégories définissent comment des situations de mélange de modèles peuvent être créées. Elles illustrent également l'influence que peuvent avoir les développeurs sur ces modèles à l'exécution. En effet, certaines techniques de programmation laissent les programmeurs décrire tous les appels aux modèles de programmation. Dans ce cas, c'est aux utilisateurs des applications de gérer tous les paramètres de ces modèles. Il est possible d'optimiser ces paramètres en fonction de l'application, de la machine, des besoins etc. Des erreurs d'optimisation sont aussi possible, car il faut gérer un très grand nombre de paramètres, et ces paramètres peuvent changer en fonction de l'architecture de la machine, etc. De l'autre cote du spectres, certains langage de programmation font une totale abstraction des modèles et gèrent les ressources sans que l'utilisateur n'ait a faire quoi que ce soit. Dans ce cas, les possibilités d'optimisation sont moindres, mais c'est aussi le cas des possibles erreurs. De ces catégories ressortent plusieurs points importants. Le premier est qu'il existe un grand nombre de techniques différentes permettant de créer des codes de calcul utilisant plusieurs modèles de programmation. Chacune a ses avantages et inconvénients, selon l'application, la machine, les utilisateurs etc. La seconde est qu'il n'existe pas de solution miracle pour toutes ces plateformes de programmation, il existe trop de modèles et techniques. De ce fait, nous avons dû dans la suite de cette thèse, nous raccrocher au bloc de base de ces modèles : ils utilisent tous des processus et processus légers. C'est le point d'entrée que nous avons utilise pour étudier et optimiser l'utilisation des ressources par tous ses différents modèles de programmation.
La seconde partie de la taxonomie définit les possibles interactions entre les modèles de programmation pendant l'exécution des simulations. Ces interactions sont découpées en deux configurations : les interactions spatiales et temporelles. Si deux modèles sont spatialement concurrents par exemple, il est possible qu'ils se disputent des ressources de calcul. Deux modèles spatialement indépendants peuvent aussi interagir entre eux, s'ils sont temporellement concurrents et qu'ils s'envoient des messages par exemple. Les configurations de mélange nous informent sur les éventuels interactions entre modèles. De plus, connaitre la configuration des modèles permet aussi de concentrer les efforts d'optimisations sur les problèmes réellement rencontrés.
Finalement, les différentes parties de la taxonomie nous informent sur les mé-

thodes de mélange de modèles ainsi que les possibles effets sur les applications. Grâce aux informations récoltés, nous savons que nous devons nous concentrer sur les processus et processus légers, mais aussi sur quel type d'erreur nous focaliser. Suite a la création de cette taxonomie, nous avons également remarque qu'aucun outil d'aide au développement ne prenait en compte le mélange de programmation. Commettre une erreur dans les paramètres des modèles ne soulève aucune erreur ou avertissement à la compilation, et il en est de même pour une mauvaise utilisation des ressources à l'exécution. De ce fait, nous avons décidé de nous pencher sur l'utilisation des ressources par les modèles de programmation.

La seconde contribution de cette thèse est une suite d'algorithmes permettant de détecter les possibles mauvaises utilisation de ressources par les processus et processus légers. Ces algorithmes restent centrés autours des problématiques liées aux modèles de programmation. Ils se basent sur des traces d'exécution, contenant des indices sur le placement des fils d'exécution sur les ressources. En utilisant ces informations, ils déterminent si certaines ressources ont pu être surchargées ou non utilisées par exemple. Nous avons implémenté ces algorithmes dans un outil d'analyse pour aider à l'optimisation des codes de calcul. La première étape est de récolter les informations sur le placement des fils d'exécution. Pour ce faire, nous avons développé une bibliothèque dynamique qui récolte toutes les informations nécessaire pendant l'exécution des applications. Cette bibliothèque apporte un très faible surcout en temps, et ne nécessite aucun changement dans le code ou la chaine de compilation de l'application ciblée. Une fois les informations récoltés, nous pouvons utiliser nos algorithmes. Ceux-ci fournissent un rapport décrivant l'utilisation des ressources par les fils d'exécution. Ce rapport permet de déterminer si un changement dans les paramètres des modèles de programmation peut être bénéfique pour les performances de l'application. Cependant, même si cet outil peut donner des indices sur les améliorations et optimisations possibles, il reste toujours un très grand nombre de paramètres à connaitre et modifier pour optimiser une application.

La troisième contribution de cette thèse est second un outil, aillant pour objectif de placer dynamique les fils d'exécution sur les ressources de calcul. Il utilise les développements et informations récoltés par ce premier outil pour déterminer dynamiquement un placement en fonction des fils d'exécution en présence ainsi que l'architecture de la machine. Cet outil n'est encore qu'une preuve de concept, mais il présente des résultats encourageants pour la suite. Nous avons tout d'abord observé que la grande majorité des codes de simulation utilisent MPI pour gérer les communications entre les nœuds de calculs. Parfois, un modèle à base de processus légers est ajouté pour optimiser l'utilisation de la mémoire partagée sur les nœuds. Dans ce type de configuration, l'optimisa-

tion la plus utilisée est d'espacer au mieux les taches MPI, et de grouper les processus légers provenant d'un même modèle. De ce fait, les taches MPI ont un maximum de ressources pour leurs calculs et les processus légers partageant de la mémoire sont regroupes sur les mêmes mémoires physiques. Nous avons utilisé cette heuristique pour développer notre outil. Les tests que nous avons effectués montrent que cette heuristique produit les meilleurs résultats sur les applications utilisées de nos jours sur les supercalculateurs. Utiliser notre outil permet de minimiser les erreurs de placement de modèles de programmation. Dans le futur, nous souhaitons développer de nouveaux algorithmes pour cet outil. Ceux-ci apporteraient une plus grande flexibilité et permettraient d'adapter notre outil à des applications présentant des comportement différents de ceux observes jusqu'à maintenant.

En conclusion, cette thèse apporte un regard sur une problématique importante du calcul haute performance : l'empilement de modèles de programmation. Avec l'évolution des architectures des supercalculateurs, cette problématique risque de devenir de plus en plus importante pour les performances des codes de calcul. Cette thèse propose une taxonomie permettant de décrire les techniques menant à l'empilement de modèles ainsi que les potentiels problèmes qui peuvent survenir à l'exécution. Nous avons également développé des algorithmes permettant de détecter les mauvaises utilisations de ressources sur les supercalculateurs. Ces algorithmes ont été implémentés dans un outil d'aide à l'optimisation. Nous avons également développé un second outil permettant de gérer dynamiquement les fils d'exécution et les ressources pendant l'exécution de codes de calcul. Toutes ces contributions sont centrées sur les modèles de programmation et prennent en compte des problématiques qui leur sont propres.

Plusieurs pistes d'amélioration sont envisagées pour nos différents outils :

- Les deux outils développés sont toujours au stade de preuve de concept. Nous souhaitons ajouter des paramètres pour leur donner une vision plus large des problèmes de gestion de ressources. Nous nous sommes pour l'instant focalisés sur les processus et processus légers ainsi que les cœurs de calcul. Il est envisageable d'étudier d'autres ressources comme la mémoire par exemple. De plus, les futures architecture risquent d'exposer de plus en plus d'hétérogénéité. Certains cœurs de calculs pourraient par exemple être spécialisés dans les communications entres nœuds. Dans ce cas, placer le fils d'exécution effectuant les communications sur le cœur approprié serait une optimisation intéressante pour les performances. Nos outils doivent donc évoluer et prendre en compte les problématiques futures.
- Nous nous sommes aussi focalisés sur un type d'application particulier. Ce type d'application correspond à la grande majorité des codes de calculs utilisés sur supercalculateurs mais cela pourrait évoluer. Dans certains cas,

l'utilisateur peut vouloir utiliser une configuration moins commune. Pour que nos outils puissent tout de même être utilisés dans ces cas particuliers, nous avons pour objectif de développer un langage simple permettant à un utilisateur de décrire ses besoins. Grâce a cette description, nos outils pourraient prendre les meilleurs décisions pour coller aux besoins des utilisateurs.

Dans un contexte plus large, cette thèse est centrée sur ce qu'il se passe quand une application utilise plusieurs modèles de programmation. Cependant, un supercalculateur fait rarement une seule simulation, plusieurs sont en cours en même temps. De ce fait, nous pensons qu'élargir les problématiques d'empilement de modèles au gestionnaire de ressource pourrait être bénéfique. Celui-ci pourrait par exemple utiliser les périodes creuse des applications pour effectuer des travaux légers. Ceci améliorerait le rendement des supercalculateurs en augmentant leur taux d'utilisation et en diminuant les pertes d'énergie.

Laboratoires d'accueil :
CEA, DAM, DIF, F-91297 Arpajon, France
Inria Bordeaux Sud-Ouest
LaBRI, Université de Bordeaux

Contents

Chapter 1

Introduction

Numerical simulation is the reproduction of real, physical behavior using computers and the tremendous computing power assiociated with them. It is used to model complex phenomena, otherwise impossible to replicate or predict. It has become an essential tool in the industry as it can reduce the cost of expensive experiments. Numerical simulation is for example present in engineering design. Automotive industry relies on simulation to design cars and perform thousands of crash-test experiments without having to build multiple expensive prototypes. The same principles apply to wind tunnel experiments performed by automotive and aerospace industry to optimize aero dynamics of a car, plane wings, or a space rocket. Oil and gaz companies reduce risks and costs when drilling for oil. Simulation also helps to predict weather forecast, for financial simulations, by biosciences, and so on. However, simulation is not only used to reduce costs in industry. It is also hugely exploited in research. Recently, reaserchers at Los Alamos Natrional Laboratory simulated an entire gene of DNA, composed of one billion atoms, that will help better understand and develolp cures for deseases like cancer [1]. Earlier, in 2012, the DEUS consortium managed to simulate the structure of the observable universe since the Big Bang [2]. Scientists from both Research and Industry perform these simulations using the most powerful computers in the world, called supercomputers. These specialised machines are designed to provide as much computing performances as possible. The science and techniques related to the design and exploitation of these computing beasts is called *High Performance Computing (HPC)*.

Supercomputers are more complex to exploit than traditional desktop computers. Two major factors are to take into account. First, supercomputers are composed of multiple inter-connected machines, and second, they are composed of high-performance processors. Let us focus on each factor independantly.

Supercomputers are composed of multiple machines. Each one is computing a part of the simulation. This means that, to deliver maximum performance, the codes need to be designed in a way that they can be split into pieces, performed in parallel. These parts sometimes need to communicate with each other to update data and move the simulation forward. All these constraints on code parallelism require new programming methods. Parallel algorithms first, but also standards to help with communications. The most widespread standard for parallel programming is called *Message Passing Interface* [3] (MPI).

Supercomputers are using high performance processors. This performance comes from the large number of compute cores. The Intel KNL is for example composed of 72 cores accessing some shared-memory. Exploiting shared-memory also requires dedicated techniques, to avoid non-deterministic results. Standards exist to help with this side of the programming too. The most widespread model for shared-memory exploitation in HPC is certainly *OpenMP* [4]. Finally, to always provide more compute power, heterogeneous components are added to already complex architectures. Some are specialised in certain kind of parallelism, for specific simulations. In the future we will certainly see heterogeneous processors, with some core specialised in communications, some in I/O operations, and so on.

Therefore, to exploit supercomputers to their maximum, simulation codes need to rely on multiple programming models standards and their implementations, called runtimes. It creates situations where multiple runtimes may cohabit and share supercomputer resources. We propose to call these situations *Runtime Stacking*. One issue is that these models were designed independently from one another. Thus, at execution time, they could potentially be unaware of each other, and compete for resources, creating resource misuse and a slow down of applications. With the complexity and specialisation of compute resources, the number of runtimes used in simulation codes is increasing, and with it the probability of resource misuse. Runtime stacking study is therefore necessary to better seize supercomputer compute power and limit performance degradations.

In this context, this thesis proposes a study of runtime staking techniques and effects on application performances. First we focus on the way runtimes can be mixed at execution time. We define runtime stacking *configurations* to describe how compute instructions from different runtimes share resources spatially and temporally. We also identify runtime stacking *categories* to illustrate the different situations where mixing multiple parallel programming models may appear. From these observations we design and implement algorithms to check the configuration of an HPC application running on a supercomputer.

These algorithms detect misuse of any resource used by the application. We then implement these algorithms in a software tool call the *Overseer* that not only checks resource usage, but also warns the user in case of misuse. This tool is used to show how runtime placement and resource usage can impact the performances of parallel applications. We then present a new approach to dynamically handle resource assignment in HPC parallel environments. This approach is implemented in the *Overmind*, a software that catches the creation of compute workers and reorganizes their binding to avoid misuse of resources relying on the previous algorithms.

This document is organized in three parts. The first one is composed of the Chapter 2 presenting the context of the thesis and Chapter 3 formulating the problem. The context chapter introduces all the basic knowledge needed to understand the document. It presents the evolution of parallel computer architectures as well as techniques to exploit them. The next chapter brings the focus on our problem with a motivating example. The second part of the thesis is composed of its contributions. We first define taxonomies of runtime stacking configurations and categories in Chapter 4. Chapter 5 then describes our algorithms to detect the current runtime-stacking configuration and check for resource usage. This chapter also presents and evaluates our implementation of these algorithms in our first tool, the Overseer. Next, Chapter 6 describes the design of our second tool, the Overmind. Finally, the last part concludes the document. We summarize our contribution and discuss future work as well as mid and long term perspectives in Chapter 7.

Part I

Context and State-of-the-Art

Chapter 2

Computer Hardware and Software Evolution

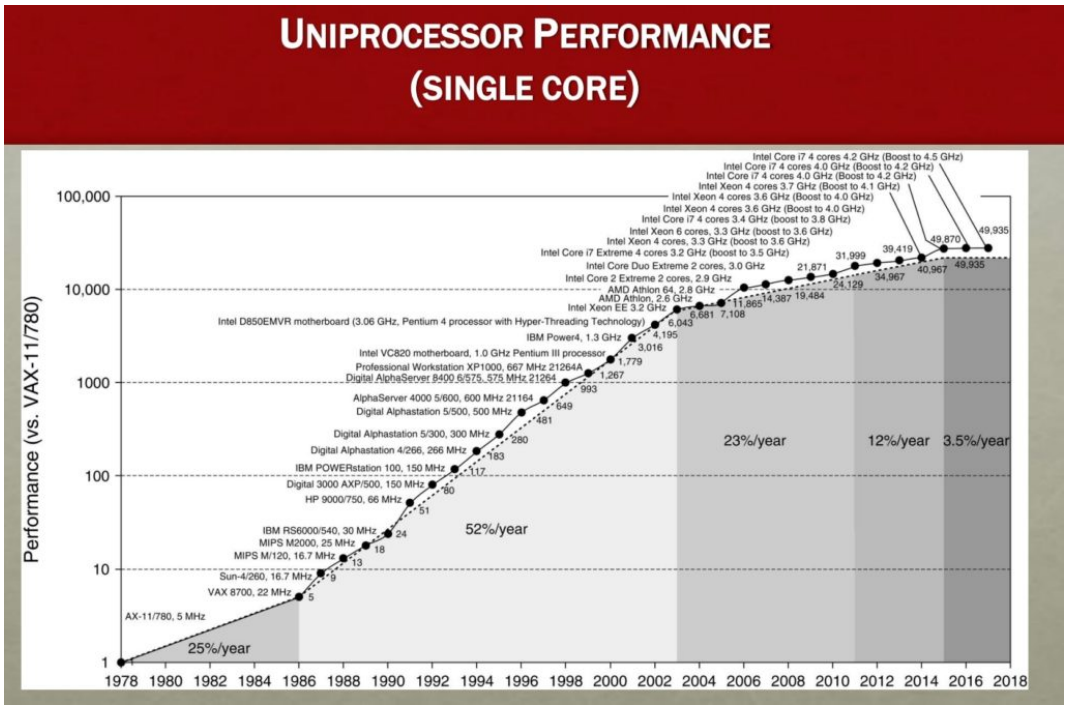
Since the invention of computers, tremendous changes have happened both in technology and design. As a comparison, a low-budget laptop bought today around $300 with any Intel i3 processor would produce around 2 GFLOPS (i.e., two billion floating-point operations per second). In 1985 the best supercomputer, the Cray-2 peaking at 1.9 GFLOPS, cost more than $10 million. This massive growth came from advances in technologies used to build computers and from innovations in computer design. This first chapter introduces basic HPC concepts needed to understand this PhD Thesis. It also presents the evolutions that took place to reach the current supercomputer performances.

Section 2.1 is an overview of important computer evolutions that led to supercomputer we use today. The technological and architectural advances are not necessarily presented chronologically as there is too much overlap and fields advancing at the same time. They are thus presented in an almost chronological, convenient order. Section 2.2 presents the architecture of current HPC clusters. Section 2.3 then presents an overview of the methods used to exploit these complex architectures.

2.1 Computer architecture: From single-core to actual multi and many-cores

This section presents the relevant advances related to computer architecture from the very beginning of computers to current massively parallel supercomputers. All the presented advances are related to microprocessors and how to exploit them. However, showing a clear chronological view of these advents is not practical. In fact, there are two main possibilities to extract more performances from microprocessors: make them work faster, or add units to process

Figure 2.1.1 – Evolution of Uniprocessor Performance since 1978 [5]



more tasks at once. Advances on both front were conducted concurrently, creating a chaotic timeline to follow. Lastly there was also another solution. This one is not to extract more from microprocessors, but from the machine in general: use more processors. This is also a massively used technique, particularly in HPC machines.

2.1.1 Moore's Law

The first design of the computer framework dates back to the early/mid 19th century by Charles Babbage. From there, and to even build the first computer, breakthroughs had to happen. In 1949, the concept of integrated circuits appeared. However, we had to wait almost 10 more years to see the first working example of such circuit in September 1958. From there, advances in technology used to build computers and innovations in computer design made possible to deliver performance improvement of about 25% per year.

In 1965 G. E. Moore predicted an increase in the number of transistors on integrated circuit doubling every year. In 1975, he revised the forecast to doubling every two years. This prediction proved accurate for several decades. Figure 2.1.1 presents the actual growth in processor performance since 1978. Performances of processors are relative to the VAX 11/780, a minicomputer introduced in October 1977, the first to implement the VAX architecture and

measured by the SPEC benchmark. We can observe the rapid growth in performances of processors from their creation to the beginning of the third millennium. Then the growth slows a bit each year. We will present the major innovations which made this rapid growth possible, and the limits we reached in the past decades.

When creating the first processors, electronic components could not fit on only one integrated circuit. Thus, multiple circuits had to be connected. The invention of the microprocessor greatly reduced the cost of processing power as well as speed up computations thanks to reduced distances between components. The first iteration of commercialized microprocessor happened in 1971 by Intel with the Intel 4004. This processor was composed of 2300 transistors. From there, the number of transistors on a microprocessor has been constantly rising based on the decreasing size of transistors. Today's processors incorporate up to 20 millions transistors. In fact, following Moore's law was possible just by decreasing semiconductor feature size. All these factors also led to a higher rate of performance improvement with an average of around 35% per year. This growth rate as well as mass-produced microprocessors led to important changes: no need for assembly language any more, and vendor independent standardized operating systems like UNIX and later Linux. This in turn made the development of a new set of architectures, with simpler instruction sets possible. The RISC architecture (Reduced Instruction Set Computer), developed in the early 80s, focused on the exploitation of instruction parallelism (initially pipeline) and the use of caches (initially simple then sophistically organised and optimized). This led to 17 years of sustained growth in performances at an annual rate of over 50% as we can see on Figure 2.1.1. This period unfortunately had to end due to multiple factors. Most importantly, process technology (the semiconductor manufacturing process) is at its limits. Indeed, diminishing semiconductor size, the backbone of performance improvement is not as easy any more, if actually possible.

Since 2003 single processor performance improvement has dropped to less than 22% per year due to the twin hurdles of maximum power dissipation of air-cooled chips and the lack of more instruction-level parallelism to exploit efficiently

2.1.2 Clock speed

Concurrently and linked to processor density evolution, another factor in performance growth is the clock rate increase in processors. In their basic operational principles, processors use transistors. These transistors put together form logical gates that can perform operations. The speed at which these operations can be done represent the clock speed of a processor. If we can give

Figure 2.1.2 – Evolution of microprocessor clock rate between 1978 and 2010 [5]

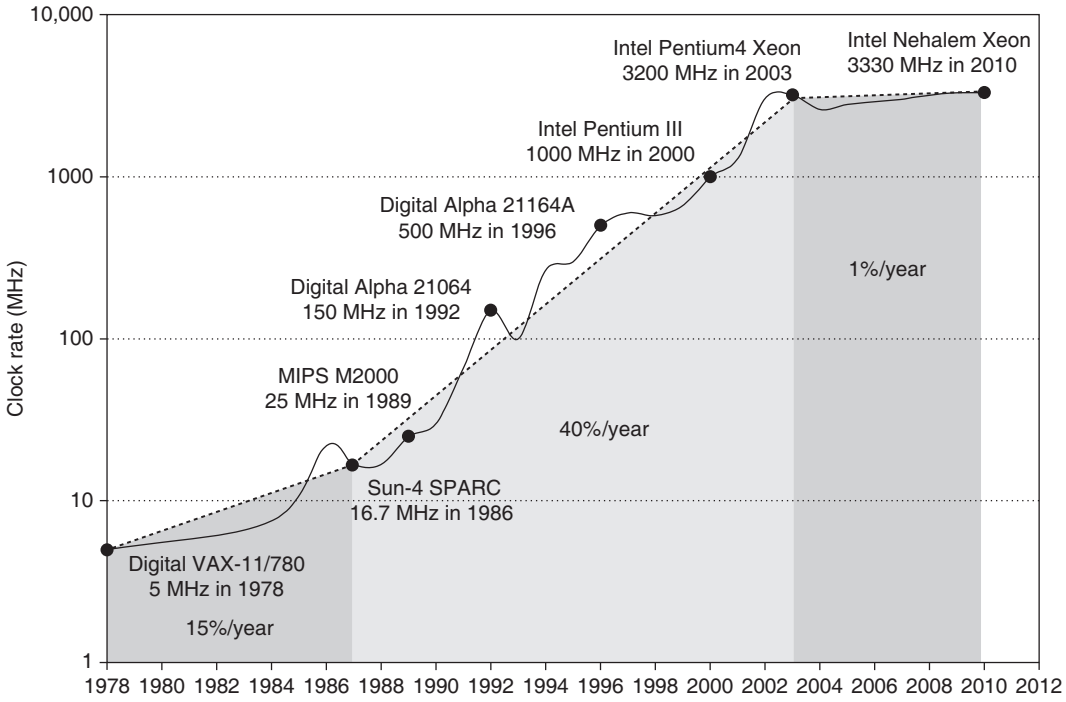
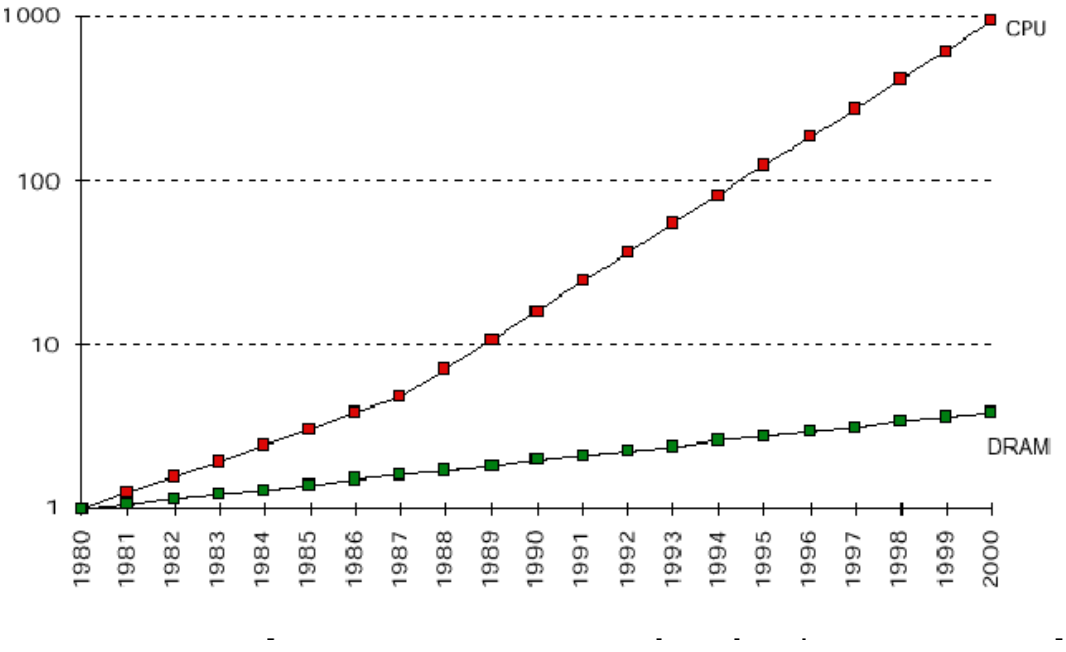


Figure 2.1.3 – Evolution of the processor-memory performance gap starting in the 1980 [6]



our processor 1 input signal and get 1 result (error free) per second, then our processor's clock rate is 1Hz. For a while, clock speed was the main concern of manufacturers. It's easy to understand why. By doubling the clock speed of a processor, codes ran two times faster. And this only by changing architecture, nothing had to be done with the code itself.

Now how can clock speed increase? Getting transistors to perform faster has a limit. This limit is the frequency at which the transistors can switch from *on* to *off* and *off* to *on*. However, a solution to this physical limit is to add more transistors, adding a new physical limit in the equation: space. We can see how Moore's law comes into account with this first solution: decreasing component size, adding more components, and in the process improving processor performances. However, since CPUs stay roughly the same size, adding more and more transistors is not possible eternally. In fact, feature size of processors is getting to its physical limit.

Moreover, energy consumption is becoming a limiting factor too. In fact power is today the biggest challenge. Two factors enter into account: getting enough power for the components and heat dissipation. If we look back at the first microprocessors, we can see that they were using less than a watt of energy. Today's desktop 3.6Gz 8 core Intel i7 is consuming 130 Watts. On the HPC side, the ARM Cavium ThunderX2, a 32 core chip running at 2.5Gz tops out at 200 Watts. Intel's Xeon Skylake line that goes up to 3.8Gz and 28 cores are consuming just over 200 Watts. This is also creating a new problem. A microprocessor is basically a 1.5 cm wide chip. This chip is heating from all the energy used. Another physical limit to take into account is the cooling that can be achieved by air. Actually this limit as long been reached. Today's HPC clusters are cooled with water. However, this technology also is reaching its limits. This led to a clock frequency evolution slow down in 2003 as we can't reduce voltage or increase power per chip. Figure 2.1.2 shows the evolution of clock since 1978 and illustrates the clock rate wall reached in 2003.

Moreover, while the microprocessor performances were improving at an almost 55% rate, the memory access time was only getting 10% better every year. Figure 2.1.3 shows this disparity. The performance gap grows exponentially, and microprocessor are becoming too powerful compared to memory. This means that the rate at which data is supplied to microprocessors is too slow. Computation is done before new data arrives, and thus precious cycles are lost waiting.

Clock rate is no longer the focal point for hardware improvements, both because of the clock rate wall and the memory performance gap. In 2004 Intel cancelled its high-performance uni-processor projects and joined others in

declaring that the road to higher performances would be via multiple processors per chip rather than faster uni-processors. This was the milestone signalling a historic switch from relying solely on instruction-level parallelism, to data-level parallelism and thread-level parallelism.

2.1.3 Parallelism Advent

As we have seen with Intel's statement, parallelism is the driving force of computer design with energy and cost being the primary constraints. The more recent Intel's Xeon processors are using up to 28 cores and 56 hardware threads. Other constructors are also following this trend. The ARM Cavium ThunderX2 for example is build with up to 32 cores and 128 hardware threads. There are two kinds of parallelism that can be exploited in applications. The fist one is data-level parallelism. Data parallelism arises when there are many data items that can be operated on at the same time. The second one is task-level parallelism Task level parallelism arises because work instances that can operate independently and in parallel are created.

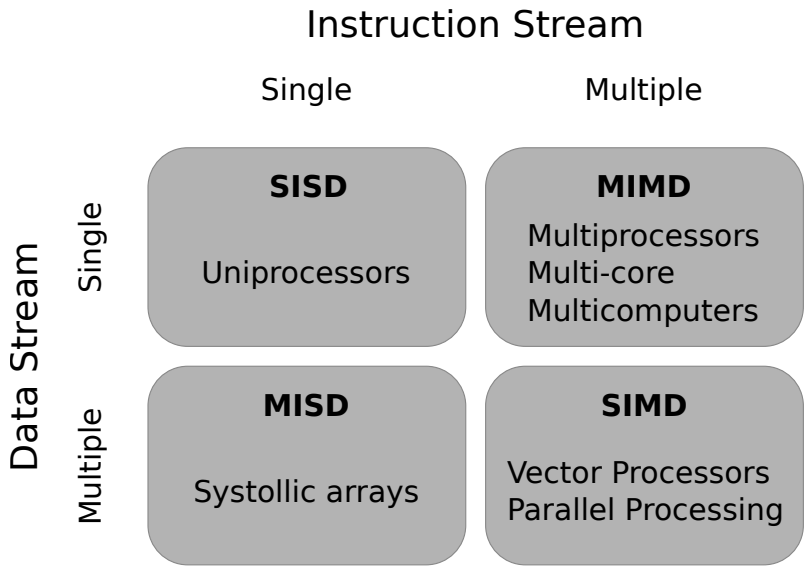
Computer hardware can exploit these two kinds of application parallelism in 4 major ways. Instruction level parallelism uses ideas like pipelining and speculative execution to exploit data-level parallelism to different degrees. A second way to exploit data level parallelism is with Vector architectures and Graphic Processor Units (GPUs) as they apply a single instruction to a collection of data in parallel. Thread level parallelism exploits either data level parallelism or task level parallelism in a tightly coupled hardware model that allows for interaction among parallel threads. Lastly, request level parallelism exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

These four ways for hardware to support the data level and task level parallelism go back to the 60s. In 1966, Michael Flynn proposed a classification, presented in Figure 2.1.4, by comparing the number of instructions and data items that are manipulated simultaneously [8]. The sequence of instruction read from memory constitutes an instruction stream. The operation performed on data in the processor constitute a data stream. This classification is still used today.

Here is a definition of each category:

- Single Instruction stream, Single Data stream (SISD): This corresponds to the uniprocessor. This category looks to be aimed at sequential computers, but they can still exploit instruction-level parallelism (pipelining, superscalar execution, speculative execution).
- Single Instruction stream, Multiple Data streams (SIMD): The same instruction is executed on multiple different data streams. Computers in

Figure 2.1.4 – Flynn's Classification of computers [7]



this category exploit data-level parallelism (vector architecture, GPUs)

- Multiple Instruction stream, Single Data stream (MISD): No commercial computer of this type exists, but this category rounds up the classification.
- Multiple Instruction stream, Multiple Data stream (MIMD): Each processor fetches its own instruction and operates on its own data. It targets task-level parallelism. This is more flexible than SIMD, thus more applicable, but also more expensive. These computers exploit thread-level parallelism where multiple cooperating threads operate in parallel. MIMD architectures are largely used in HPC clusters, and exploit request-level parallelism where many independent tasks can proceed in parallel. This category often involves little to no need for synchronization.

This taxonomy is a coarse model as many processors are hybrid of these four categories. Still, it is useful to create a loose classification and describe processors, architectures and computers. The following subsections present techniques used to implement these concepts.

2.1.4 Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) is a form of parallelism which aims at executing multiple instructions simultaneously. It can be implemented in multitude ways and as been present for almost as long as processor exists. It has been implemented in superscalar processors. The Cray CDC 6600 which dates back to 1966 is often mentioned as the first superscalar design. Pipelining for example is used in all processors since about 1985. Other techniques extend basic pipelining concepts by increasing the amount of parallelism exploited

among instructions. There are two approaches to Instruction Level Parallelism, one that relies on hardware to exploit parallelism dynamically, and one that relies on software to find parallelism statically at compile time. Let us first look at pipelining.

2.1.4.1 Pipelining

To access the computer's computing resources, human beings need to write computer programs. These programs are transformed into instructions that a processor can execute. An instruction is divided into multiple actions. A simplified example would be adding two numbers together. First we need to pull the two numbers from memory and place them in the processor, then perform the addition, and finally push the results back to memory. There were three steps here: getting from memory, the actual addition instruction, and the save into memory. Each of these steps are done by different parts of the processor, thus could be done at the same time. Of course, it is not possible to store the result of the addition before its completion, so it is not possible to do everything at the same time. However, if multiple operations are queued, all the steps can work at the same time on different instructions, as machine working on an assembly line. This is what pipelining is, breaking instructions into smaller parts to create an assembly line and work on multiple instructions in parallel. If the step times are perfectly balanced, the time per instruction on a pipelined processor is equal to '*time per instrucion / number of stages*'. In practice, steps are not balanced, and the stage time is equal to the longest step. This means that pipelining involves some overhead. In fact, the time to complete an instruction on a pipelined processor is not its minimum possible, but if multiple instructions are queued, the overall time will be lowered. The pipeline increases the instruction throughput making programs run faster even though no single instruction runs faster. Pipelining exploits parallelism among instructions in a sequential stream. This has the large advantage to be invisible to the programmer, as well as to be largely applicable.

To illustrate pipelining, we present the classic five-stage RISC pipeline displayed in Figure 2.1.5. First we need to look at the RISC instruction set. The basic operations are: ALU instructions, Load and Store, Branch and Jumps. Each instruction in this RISC instruction set can be implemented in at most 5 clock cycles. The 5 clock cycles presented in Figure 2.1.5 are as follows: Instruction Fetch, Instruction Decode, EXecution, MEMory access, Write-Back. Note that branch instructions require 2 cycles, store instruction 4, and all other instructions 5. Although each instruction takes 5 clock cycles to complete, the hardware will be executing some part of the different instructions each cycle, resulting in the execution of an instruction per cycle. From the figure, we see at the first time step the instruction A entering the pipeline:

Figure 2.1.5 – Basic RISC five-stage pipeline [5]

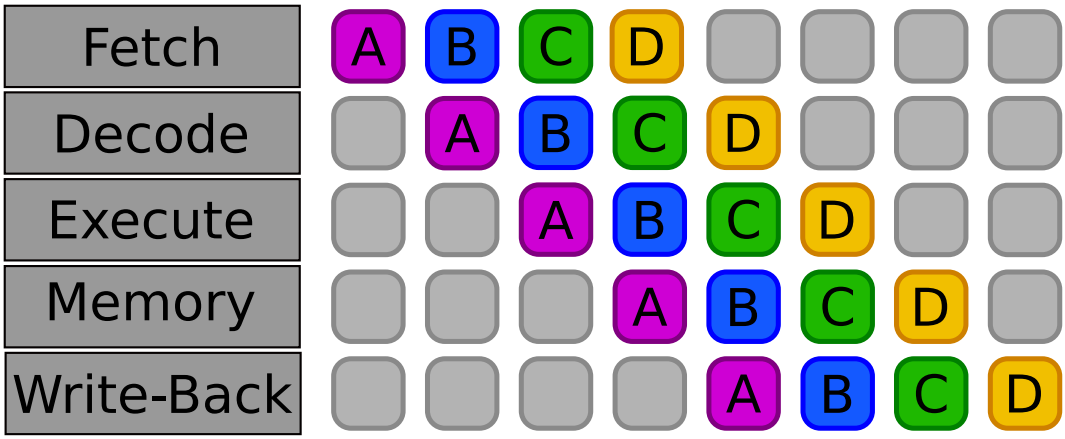
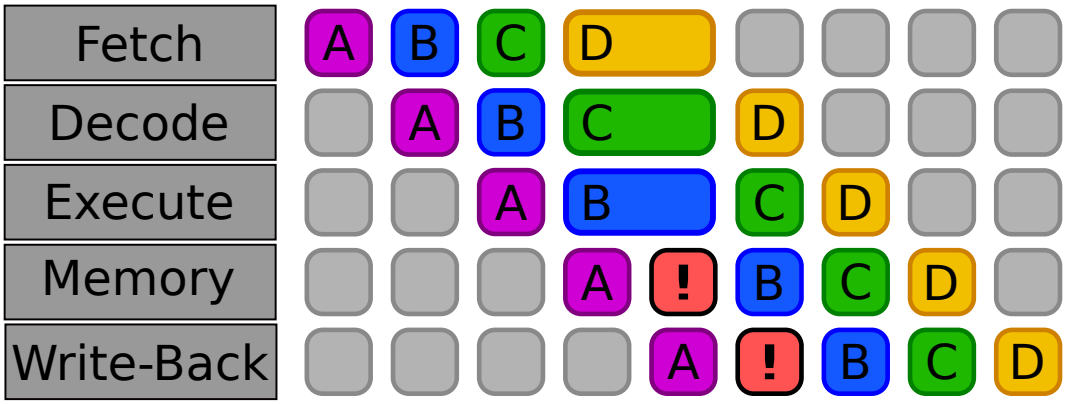


Figure 2.1.6 – Basic five-stage pipeline with a bubble starting at the instruction decode phase



the instruction is fetched from memory. At the second step, instruction A is decoded while instruction B enters the pipeline and gets fetched. By adding more instructions in the pipeline every time step, we can hope to completely fill the pipeline. Unfortunately, blindly pipelining instructions is not always the best thing to do. Pipelining hazards can happen. Those are data hazards and control hazards. They can sometimes stall the pipeline, these situations are called pipeline 'bubbles'. To solve these problems other instruction level parallelism mechanisms can be used.

2.1.4.2 Hazards in pipelines

Let us look at some usual hazards in pipeline and ways to solve them.

Data Hazards happen when an instruction in the pipeline is dependent on the result of another instruction in the pipeline. Consider two instructions: an addition followed by a subtraction. We represent this sequence as follows:

```
ADD R1, R2, R3;
SUB R4, R1, R5;
```

ADD R1, R2, R3; is the addition operation, where the data in register *R2* and *R3* are summed and the result is stored in *R1*. In the same way, *SUB R4, R1, R5;* is the subtraction operation, subtracting the data in register *R1* to data in the register *R5* and storing the result in *R4*. The subtraction is using the result of the addition (the *R1* register). If we look back at the pipeline from figure 2.1.5 we can see that to perform the *ID* stage, the *SUB* needs to wait for the *WB* stage of the *ADD* operation. This creates a 'bubble' in the pipeline i.e., a time loss for all instructions. Figure 2.1.6 shows the creation of a bubble at the 'decode' phase. This simple stall can be avoided by using a hardware technique called forwarding (also called bypassing). Indeed, the result of the *ADD* operation is not really needed until it is actually produced. It can be directly used for the next instruction (the *SUB*) without going through the *MEM* and *WB* phases. Thus, in the end of the *EX* phase, a copy of the result is made and fed to the ALU for the next instruction in case of a dependence between two instructions. Note that the *MEM* and *WB* phase are still completed after the *EX* phase as usual.

Unfortunately, bypassing cannot solve all the data hazards. Consider the following sequence:

```
LD R1, R2;
ADD R3, R1, R4;
```

The *LD* instruction loads data from memory to processor registers. This instruction cannot forward the data until the *MEM* phase. Thus, it would be too late to avoid a stall. A solution to this problem would be to insert another independent instruction between the Load and the Add, thus filling the pipeline. Of course this instruction should not create a stall for it to be a good candidate. This technique is called out-of-order execution or dynamic scheduling. It avoids delays that occur when the data needed to perform an operation are unavailable.

Control Hazards , also called branching hazards, occur when a branch can be taken in a program, for example after an 'if' statement. When there are two possibilities, the processor will not know the outcome of the branch before it is computed, and thus will not be able to insert new instructions in the pipeline. There are many methods for dealing with pipeline stalls caused by branch delays. There are for example multiple prediction schemes [9] which look at the most likely way the branch should go and fill the pipeline with these instructions. For example, in a *while* loop, a scheme could say 'always assume the while will continue for another iteration'. This will, most of the time, work better than a random scheme as we can assume that if the programmer used a 'while' loop, the following instructions should (hopefully for the pipeline) be performed more than one time.

A 'recent' technologies called SMT can also help reduce hazards in pipelines. It was first researched by IBM in 1968. However, the first commercial modern desktop processor to implement SMT was the Intel Pentium 4 released in 2002. It is now included in most of the Intel processor line. This technology is better known under its Intel denomination of 'Hyperthreading'. Other constructors will follow with sometimes new denomination. IBM releases the POWER5 in 2004 which includes a two-thread SMT engine, Sun Microsystems release the UltraSPARC T1 'Rock' in late 2005 which uses its own 'CMT' approach, and so on. The principle is simple: a hyperthreaded core will be executing two (or more) material threads at the same time. However, these threads will be sharing the core's ALU components. On the physical core, only data and control registers are duplicated. Thus, two hyperthreads also share caches and pipeline. In the end, hyperthreads are helping ILP by filling pipeline hazards. However, two hyperthreads does not mean twice the performances as ALU is not duplicated. At the Pentium 4 release, Intel claimed that the hyperthreaded version added up to a 30% speed improvement.

2.1.4.3 Conclusion on Instruction Level Parallelism

At the beginning of the third millennium, research showed that Instruction-Level Parallelism was at its limit. Indeed, processor were beginning to get

inefficient in terms of performance per watt. Moreover, the complexity of the processors was getting to high, increasing the issue rate. By 2005 the focus shifted to multicore processors. Thread level parallelism would increase performance, and Instruction Level Parallelism would not be the main focus any more. During the same period, SIMD and data-level parallelism saw developments. The next sections introduce Thread-level parallelism as well as Data-level parallelism.

2.1.5 Thread level parallelism

To exploit thread-level parallelism, the MIMD model is often used. It is the architecture of choice for general-purpose (multi-)processors. Indeed, MIMD offers flexibility as it can be used for a single core processor, an application using multiple threads running on a multicore processor as well as for multiple tasks from multiple applications running at the same time. Moreover, MIMD multi-processors can be build efficiently by replicating single processors.

2.1.5.1 Multicore Architectures

As we've seen, the improvement of computer performances happened through shrinking integrated circuits. Starting in the 1990s, technology allowed processor designers to place multiple microprocessors on a chip. Each processor is then called a core. Initially called on-chip multiprocessor or single-chip multiprocessor, processor including multiple 'cores' are now called multicore. Today they are used in almost all personal computers as well as HPC clusters and in other application domains like embedded, network and so on.

As the clock rate limit was getting closer, increased use of parallelism and thus multicore has been the main research interest to improve overall processing performances. Cores on these processors usually share some resources like second- or third-level caches, or I/O buses for example. These processors are built to exploit the MIMD class of computer parallelism. Each core is executing its own instruction stream. These could be used in general purpose computer to run multiple applications at the same time with more fluidity than with only one core scheduling multiple applications. They can also be used to run multiple threads of the same program as we will see later. Actually, multicore can also be used to exploit data-level parallelism. However, they would certainly be slower than dedicated SIMD machine optimized for these operations. As we will see in a future section, SIMD exploitation requires some prerequisites to be optimal. However, taking advantage of a multicore with n cores is usually done with n threads executing the same instructions at the same time.

Possible gains are limited by the parallelism that a software can generate

as described by Amdahl's law presented at the AFIPS Spring Joint Computer Conference in 1967 [10]. These performance gains are called speedup. Amdahl's law gives the theoretical speedup of a fixed workload that can be expected when system's resources are improved. It is often used in parallel computing to predict speedup when using more compute nodes or cores for an application, and to determine efficiency of adding more compute resources. Here is the theoretical speedup formula:

```
1/(1-p)
```

where p is the part of the workload that can be parallelised. Basically a workload's speedup is limited by its non-parallel part. By adding an infinite number of resources to a perfectly parallel workload we could reach almost instant computation. However, if the non parallelised part is one hour long, then the resulting execution will be one hour long. Figure 2.1.7 shows the evolution of the theoretical speedup in function of the number of processors. Using the efficiency formula: $E = S/p$ where E is efficiency, S is the speedup gained by a hardware upgrade, and p the parallelism of the target workload, we can determine if making a hardware upgrade is efficient enough or not. For example, it is possible to study the efficiency of making an upgrade compared to its cost.

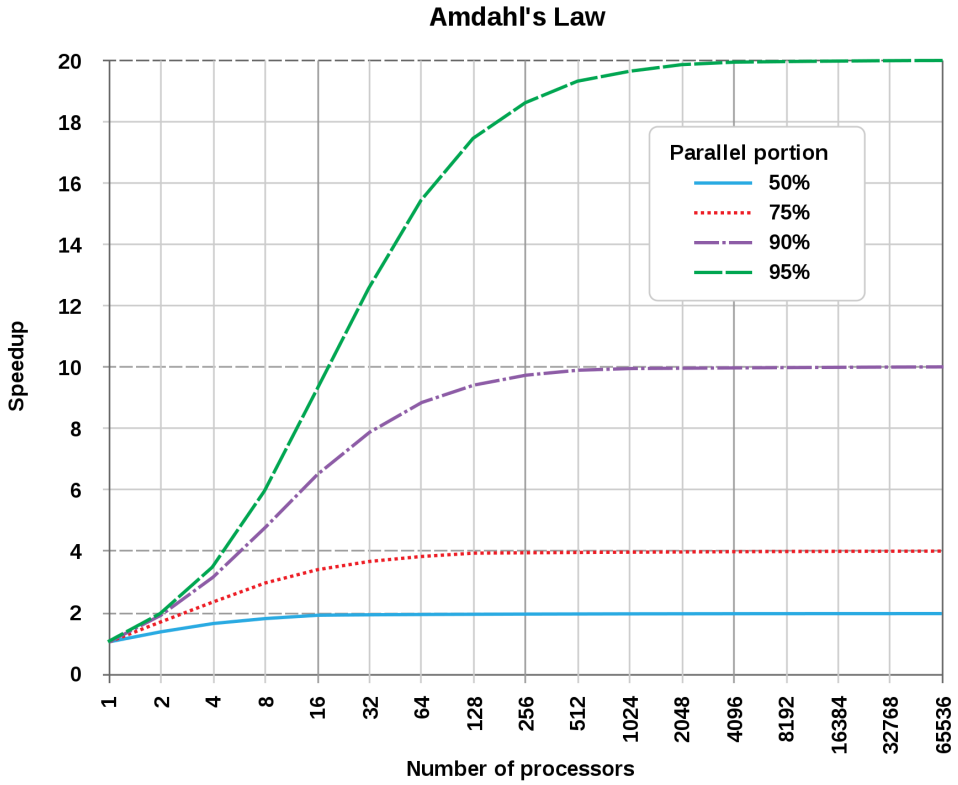
However, the number of cores is not the only factor to take into account. The raw material for a processor to work is data. This data is stored in memory and needs to find its way to the processor's registers.

2.1.5.2 Memory Hierarchy

Data are stored in memory. Obviously the best possible memory would be unlimited and fast. The first factor's limitation is obvious: memory takes space on chips, thus it is not possible to fit unlimited memory there. It is possible to get a very large amount of memory a little further away from the processor, but then we affect the second factor as we get higher latency than close on-chip memory. A clever solution to these two factors is *memory hierarchy*.

The *locality of reference* or *principle of locality* tells us that an application as the tendency to access the same set of memory locations repetitively over a short period of time. This is due to the way codes are written, using loops and arrays and updating the same set of data over and over. In fact, widely held rule of thumb is that a program spends 90% of its time in 10% of the code. This 10% are loops, accessing the same data multiple time or contiguous data objects that can be fetched by batch. Then we can differentiate between two types of reference locality. Temporal locality, the first one, refers to the use of the same data within a small duration time. The second, spacial locality,

Figure 2.1.7 – Amdahl's law: evolution of the theoretical speedup [10]



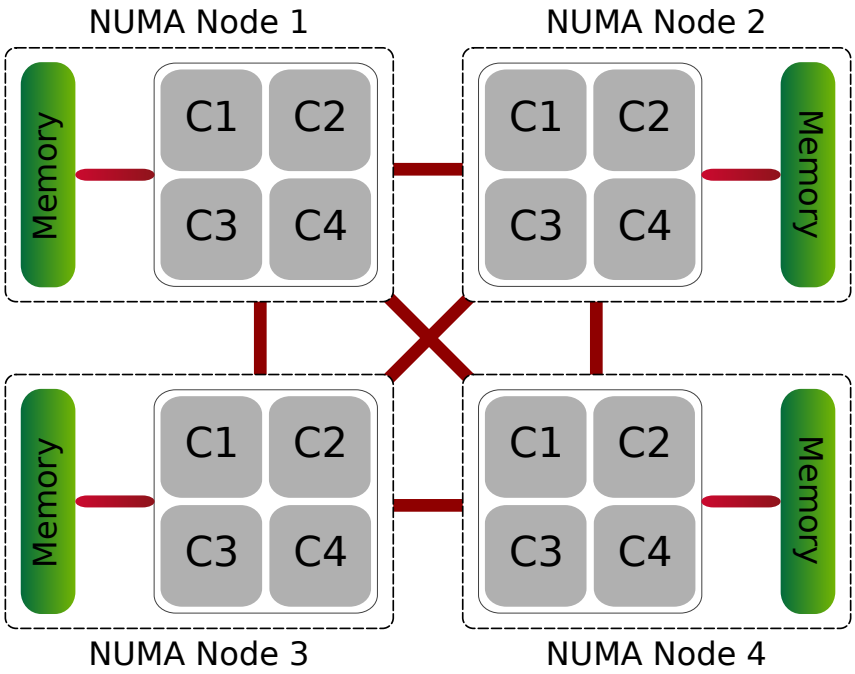
refers to the use of data elements close to one another. There is also a special case of spacial locality called sequential locality where arranged data elements are accessed linearly, such as going through an array. This principle of locality (as well as the cost of large and fast memory) led to hierarchies of memory of different sizes and speeds.

Memory is organized into a hierarchy of several levels where each level is smaller, faster, and more costly per bit than the previous one. The objective is to provide a memory system with speed almost as fast as the last level, but with cost and size of the first. Here are the main levels present in almost every memory hierarchy.

- Registers: The smallest memory on the processor. Registers may hold an instruction, a storage address, or any kind of data. Usually one word or a couple words of data in vectorial processors. The processor units (the ALU) directly use data contained in registers to perform computations. This is the fastest memory, directly accessed by the processor, present in very limited amount.
- Caches: This memory is really close to the cores. It is actually embedded in microprocessors and thus is extremely quick to access. However, as space is limited on the chip, cache memory is really small. They range from tens of KB to a hundred MB of data.
- Main memory: Main memory provides data to caches and serves as I/O interface. It is also called RAM (Random Access Memory). Modern compute node typically have around 256 GB of memory per node.
- Disks: really large storage devices. They are used to store the machine's file system. Depending on usage, these disks can be as fast as SSD and as slow as LTO (Linear Tape-Open). As multiple disk can be used, capacity is almost unlimited.

Most CPUs have multiple caches. First, a hierarchy of instruction caches. These caches contain the instructions to be executed by the processor. Then a second hierarchy, this time containing data. We often talk of cache levels (L1, L2, L3 and so on). Usually with multicore processors, each core has access to its own L1 and L2 caches. The first level is often split for instruction (L1i) and data (L1d). The L2 cache is only used for data. Then the L3 cache is shared between all cores. L4 and higher cache level are highly uncommon. Usually, L1 cache is only a couple dozen of KB per core. L2 is a bit larger with a few hundreds KB per core. L3 would be between a hundred MB and a GB per chip. As caches are tiny, programs are loaded into main memory to be executed. Main memory is significantly larger than caches with up to hundreds of GB (and reaching the TB). However, it is also significantly slower. The order of magnitude to read data is 1 cycle to read a register, less than 5 to read in L1 cache, 10 to 15 to read in L2 and less than 100 to read in L3. Reading data from main memory is around 200 to 400 cycles. Lastly, at the

Figure 2.1.8 – Diagram of a basic four node NUMA system



end of the hierarchy we can find disks. These are really slow, even compared to RAM memory. Access to data in disks could cost a dozen ms (where RAM access is around 100ns and L1 cache less than 1ns). However, they are really cheap compared to the other memories and thus offer almost unlimited storage. Actually, tape data storage is used to store large amounts of data that don't need to be read often. This technology is useful for its cheap price, and low power consumption as tapes don't need to be powered. Accessing data stored on disks takes a few seconds to minutes depending on the technology used.

2.1.5.3 Non Uniform Memory Access (NUMA)

An additional factor to take into account when looking at memory access time is distance from each core to the memory. Indeed, two memories from the same hierarchy level can have different access times. This is the case on NUMA systems which are often used in HPC clusters. Under the NUMA memory design, a processor can access its own local memory faster than non-local memory. The non-local memory could be memory local to another processor or memory shared between processors. Figure 2.1.8 is the representation of a simple 4 processors NUMA node. Multiple cost-effective nodes on different sockets are connected by a high-performance connection. Each node contains processors and memory. However, a node is allowed to use memory from all other nodes thanks to a memory controller that keeps memory coherency. Access time to remote memory will obviously be slower than local memory. This

is called the NUMA effect. This means that using memory efficiently takes some thinking but the upside is in limiting the number of message exchange. Indeed, NUMA effect between two cores on the same NUMA system will certainly be less time-consuming than the cost of messages sent on the network.

2.1.6 Data level parallelism in vector and manycores

There are two main variations around SIMD: vector architectures (actually designed in the 70s), and graphics processing units (GPUs). As SIMD will not be a focal point of this thesis, this section only highlights the main principles of these SIMD variations.

2.1.6.1 Vector Architecture

Vector processors (also called array processors) are designed to operate on multiple data at the time. Vectors are one-dimensional arrays of data. With a single instruction, the processor will operate on all the data contained in the vector. Data are then put back into memory. The issue is to get data from memory to the processor. Indeed, filling the vector requires a large amount of data. To reduce the time consumed by load and stores, these steps are deeply pipelined. In the end, working on a lot of data at the same time with one instruction (i.e., using large vectors) helps hiding the memory latency. However, large vector means specific set of data, with no data dependencies or hazards. This also means that when parallelism is not optimized for vectors (i.e., when using instruction on small vectors), this kind of processors becomes less efficient.

2.1.6.2 Manycore architectures

Manycore processors are architectures using a lot more cores than traditional multicore processors. They are composed of simpler cores, worst at single thread performances but optimized for higher throughput and/or lower consumption. They benefit from high degrees of parallelism. In fact, using these architectures for MIMD would be highly inefficient. Moreover, to limit energy consumption and provide more space for cores, these chips usually do not include out-of-order execution technology, deep pipelines, and large caches. There exists many specific manycore architectures. During this thesis, I used the Intel Xeon Phi, which has MIC (Many Integrated Cores) architecture. The first commercialized in 2013, called Intel Knights Corner (KNC), was a coprocessor using 57 to 61 cores. Each core was using 4 hyperthreads. The version I used during this PhD is the Intel Knights Landing (KNL), commercialized in 2016. This one is composed of 64 to 72 cores, with still 4 hyperthreads per core. On the other hand, another kind of manycore are GPUs (Graphic Processing Unit). These can be described as manycore vector processors. Originally designed to

rapidly manipulate and alter memory to create images intended for output to a display, they have been redesigned to be used in HPC clusters. Thus, all the cores can only execute simple instructions. Moreover, at each clock tick, every core is executing the same instruction (on different data). This highly parallel structure makes them really efficient for algorithms that process large block of data in parallel. In HPC, this is particularly interesting for matrix computations used in simulations.

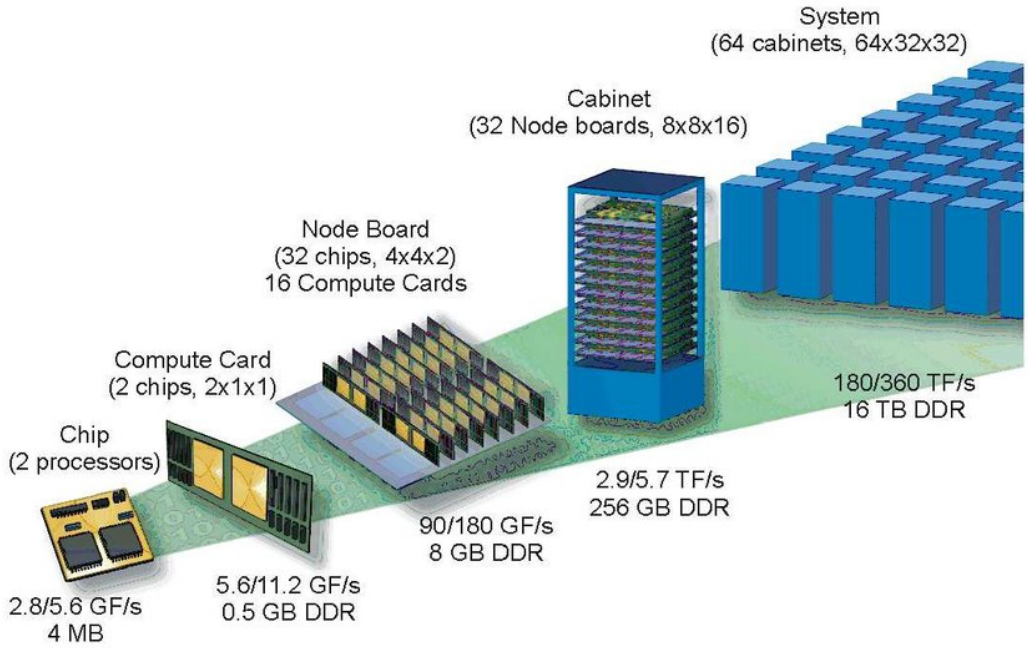
2.1.7 Conclusion on evolutions

Evolutions presented are not the only ones that permitted computer performance growth rate. Other evolutions in memory, cache, software also helped. These evolutions are not presented here as they don't add that much to the point we are trying to make: architectures are becoming complex, with a lot of processors, cores, hardware threads, deep memory hierarchies and so on. This complexity brought better possible performances, but also more complex programs and optimizations. The next sections present the actual look of HPC clusters architecture as well as techniques used to harness their compute power.

2.2 Supercomputer Architecture

Simulations are often iterative calculations, each step refining the solution. The closer the computed solution is to the measured value, the more accurate the solution is. The objectives of current supercomputer is to compute fast and accurate simulations. The next global goal is to reach the Exascale milestone. To do so, they use all the resources at their disposal. As we have seen, architectures are always evolving and supercomputers use the best innovations among them. Today in HPC, supercomputers are massively parallel systems. Figure 2.2.1 shows how supercomputers can be assembled. They are composed of a large quantity of nodes linked together. A node is itself composed of multiple multicore processors, and sometimes also connected to accelerators like GPUs. Each node uses its own operating system. In the end, programming applications for these architectures is a complex task. Programmers need to create highly parallel code taking clusters' topology into account. This means code capable of using a large quantity of nodes and cores, accessing memory efficiently, not wasting time in communication and so on. Moreover, each generation of supercomputer adds a layer of complexity and the exascale machine will, without a doubt, provide even larger problems to the code developers [12].

Figure 2.2.1 – Architecture of the Blue Gene/L Supercomputer [11]



2.2.1 Supercomputers Tops

Before we can look at how to use these complex architecture, let us look at the current most powerful computer systems in the world. A couple of tops exist to determine which computer is the 'best'. In HPC the Top500 [13] is the most used one. Twice a year, machines are compared on a specific benchmark and the best 500 are ranked. This benchmark is the `Linpack benchmark` [14]. It computes the solution of a dense linear system with n equations and n unknown. In the end, it determines the Flops of the computer (i.e., the number of floating-point operation it can compute per second).

Table 2.2.1 shows the last Top500 highest ranked computers (results from the June 2019 Top500 ranking). We can see that the United States dominates the ranking with the top two places, as well as 5 computers in the top 10. China is not far behind with two computers at the third and fourth places. France's first computer to appear in the Top500 is PANGEA III, an industrial machine from the Total firm. It is also the most powerful industrial computer of the Top500. The first research computer from France, Tera-1000-2 from the CEA places as sixteenth. However, the Flops metric does not entirely reflect HPC applications. While it is true that some codes do a lot of matrix computations that are highly parallel, some don't.

Rank	Previous Rank	Name	Country	Total Cores	Accelerator/Co-Processor	Rmax [TFlop/s]
1	1	Summit	United States	2414592	2211840	148600
2	2	Sierra	United States	1572480	1382400	94640
3	3	Sunway TaihuLight	China	10649600		93014.59388
4	4	Tianhe-2A	China	4981760	4554752	61444.5
5		Frontera	United States	448448		23516.4
6	5	Piz Daint	Switzerland	387872	319424	21230
7	6	Trinity	United States	979072		20158.7
8	7	ABCI	Japan	391680	348160	19880
9	8	SuperMUC-NG	Germany	305856		19476.6
10	11	Lassen	United States	288288	253440	18200
11		PANGEA III	France	291024	270720	17860
18	16	Tera-1000-2	France	561408		11965.5

Table 2.2.1 – Top 10 best ranked computer in the Top500 plus the fist two French ones

Thus, other benchmarks and tests are proposed. In particular, the High Performance Conjugate Gradient (HPCG) benchmark, which uses the Conjugate Gradient algorithm, is intended to complement the Linpack benchmark. Table 2.2.2 shows the top computers ranked with this benchmark (results from June 2019). The United States are also well represented in this top with the top two places and 4 computers in the top 10. We can also see that Summit and Sierra are the top two computers of the two rankings. The third computer in this top is the K computer from Japan which was 20th in the Top500. This shows that the two rankings and benchmarks look at different optimizations made in computers architectures. Tera-1000-2, the French supercomputer from CEA, is just outside the top 10 with the 11th place. But performance is not the only issue any more, energy consumption is also a limiting factor for supercomputers. It is now at the core of architectures innovations.

To reflect this focus, another ranking can be used. It is called the Green500, and ranks computers from the Top500 depending on their energy consumption (or GFlops/watt). Table 2.2.3 shows the results of the last Green500 ranking (results from June 2019). This time Japan and the United States are dominating the top 10 with 3 computers each and the four top places. We can see that a couple of top 10 computers from the Top500 are still well places here. Summit and Sierra, the first two computers of the Top500 (and HPCG) from the United States as well as ABCI the 7th from Japan. The fact that Sierra and Summit appear in the top 10 of these three rankings shows that the new supercomputers look for low energy consumption to reach higher performances.

2.3 Parallel programming models

We have seen that HPC architectures are becoming more and more complex. To exhibit more parallelism, more cores are added, but memory per core is

Rank	Top500 Rank	Name	Country	HPCG [PFlop/s]
1	1	Summit	United States	2.926
2	2	Sierra	United States	1.796
3	20	K computer	Japan	0.603
4	7	Trinity	United States	0.546
5	8	ABCI	Japan	0.509
6	6	Piz Daint	Switzerland	0.497
7	3	Sunway TaihuLight	China	0.481
8	15	Nurion	Korea	0.391
9	16	Oakforest-PACS	Japan	0.385
10	14	Cori	United States	0.355
11	18	Tera-1000-2	France	0.334

Table 2.2.2 – Top 10 best ranked computer with HPCG Benchmark plus the first French one

Rank	Top500 Rank	Name	Country	Total Cores	Rmax [TFlop/s]	Power (kW)	Power Efficiency [GFlops/Watts]
1	472	Shoubu system B	Japan	953280	1063.305	60.4	17.604
2	470	DGX SaturnV Volta	United States	22440	1070	97	15.113
3	1	Summit	United States	2414592	148600	10096	14.719
4	8	ABCI	Japan	391680	19880	1649.25	14.423
5	394	MareNostrum P9 CTE	Spain	18360	1145	81.03	14.131
6	25	TSUBAME3.0	Japan	135828	8125	792.08	13.704
7	11	PANGEA III	France	291024	17850	1367	13.065
8	2	Sierra	United States	1572480	94640	7438.28	12.723
9	43	Advanced Computing System (PreE)	China	163840	4325	380	11.382
10	23	Taiwania 2	Taiwan	170352	9000	797.54	11.285
33	47	JOLIOT-CURIE SKL	France	79488	4065.55	917	4.434

Table 2.2.3 – Top 10 best ranked computer in the Green500 plus CEA's best one

getting smaller. To limit energy consumption, co-processors are added to the already complex multiprocessors nodes. In the end, machines are made of nodes with dedicated memories linked together by a network. Each node is then composed of multicore processors with their own memory but also sharing collective memory with each other. Co-processor and accelerators like GPUs can also be added to the mix. To exploit all these resources, application developers need to create highly parallel codes, by taking complex architectures into account. To minimize the complexity of programming on these clusters, programming models have emerged. They are an abstraction of the parallel computer architecture used to express parallel algorithms. They are evaluated on there generality (how many problems they can express and how many architectures can use them), and performances. This section presents the main ones and their basic mechanisms. Programming models are separated into two big groups: shared-memory models to help manage memory shared by multiple cores of the same node, and distributed memory models that are used to send messages between cores or nodes all over the machine.

2.3.1 Shared-memory models

Processors are now composed of multiple cores. All the cores of a processor have access to its memory. Moreover, cluster nodes are often composed of multiple processors creating NUMA nodes. These nodes also have memory shared by all the processors. Using a shared memory programming model on these kinds of architectures allows multiple threads or processes running on the same processor or node to share data and communicate. Threads read and write asynchronously in shared memory. Asynchronous access to memory can lead to race conditions when multiple threads read and write in the same location. Specific techniques are used to avoid these hazardous behaviours. Programmers can implement locks, mutex, semaphores and in a larger extent critical sections to protect data from race conditions. All these mechanisms can be used to 'protect' variable from concurrent accesses and thus certify memory coherence. If a multi-threaded code only manipulates shared data structures so that all threads can't have unintended interactions it is said thread safe.
We then present the two most commonly used shared programming models in HPC, POSIX threads and OpenMP.

2.3.1.1 POSIX Threads

POSIX Threads, usually referenced to as **pthreads**, is part of the POSIX (Portable Operating System Interface) standard. This standard specified by the IEEE Computer Society is used to maintain compatibility between operating systems. The POSIX Thread part defines an API to create, control multiple flows of work that overlap in time. These flows are called threads. To understand

what a thread is, we first need to define the process. A process is an instance of a running program. It is created by the operating system and stores all the resources the program needs like memory, registers, program counter, as well as identification (program, user, ...), and so on. Figure 2.3.1 is a simplified view of a process memory layout. The first part (bottom part) contains the program code, static and global variables and dynamic variables. Then the `heap` contains all the memory allocated at runtime by the malloc family functions. On the other side the `stack` contains function parameters and return adresses, as well as non-static local variables. Between the `heap` and the `stack` lies free memory that can be used by the program at runtime. This memory is used by growing either the `heap` (towards the `stack`) or the `stack` (towards the `heap`). In a multithreaded process, threads share all this memory. When a thread is created, a new stack is created in the process memory. This memory contains the thread program counter, registers, state and his own stack. As threads share resources with the process, their creation is often faster than the creation of a process. Similarly, switching between threads is faster for the operating system than switching between processes. These advantages make threads attractive for performance, even more so when a multithreaded process can use multiple cores from a process or a node. However, using POSIX threads requires attention as sharing data may lead to non thread safe programs. POSIX threads are still used in many (if not all) libraries using threads (like OpenMP), as they are compatible with all POSIX systems.

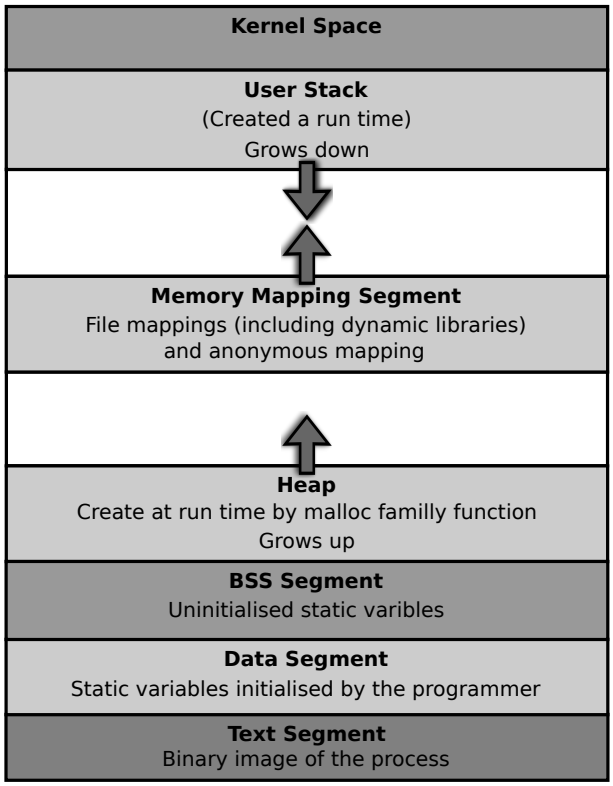


Figure 2.3.1 – Process memory mapping

2.3.1.2 OpenMP standard

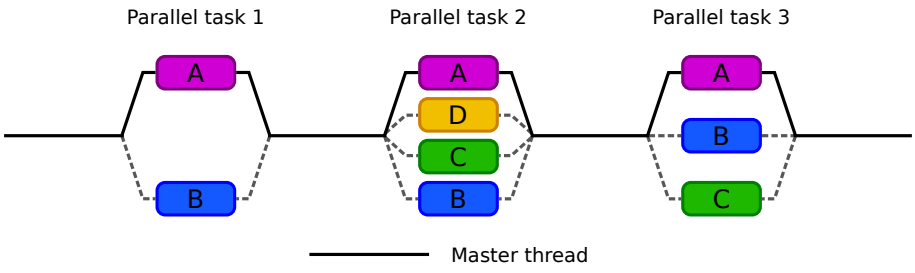


Figure 2.3.2 – Representation of the fork-join model [15]

```
#pragma omp parallel for
    for (i = 0; i < 10; i++) {
        c[i] = a[i] + b[i];
    }
```

Listing 2.1 – A loop parallelised with OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) dedicated to parallel computing. Its utilisation is simpler than the POSIX threads as the API consists of a set of compiler directives and a library that influence runtime behaviour. OpenMP is build around the fork-join model of parallelism presented in Figure 2.3.2. This model works as follows: a master thread is created at the start of the application. Then, when needed, new threads are created to accomplish various tasks. With OpenMP for example, it is straightforward to add parallelism to a loop. This is illustrated by the code snippet in Figure 2.1. Note that the only difference between the sequential and the parallelised version of this loop is the `pragma` directive on the first line. Actually, if OpenMP is not supported at compile time, the OpenMP pragma will be ignored and the code will still compile without error, and execute sequentially. With the parallel version, OpenMP threads will be assigned an independent set of iterations. These threads will work in parallel to perform the actions in the loop.

OpenMP is widely used in HPC applications for its ease of use and compatibility with simulation codes that rely on loops to iterate along the simulation. OpenMP also allows other types of parallelism, still based on threads, like task parallelism (introduced in May 2008, in the OpenMP 3.0 specifications). Tasks are used when an algorithm is creating different work tasks to execute. Some tasks are dependent on the other's results, thus they cannot be executed in any order. A scheduling algorithm is used to certify task order and results coherency.

2.3.1.3 Shared-memory models conclusion

Shared memory models are extremely important to harness multicore processors and multiprocessor nodes. However, for larger scale machines, only using shared memory is not sufficient. Indeed, when using multiple nodes for a simulation, we need to be able to exchange information between nodes. To accomplish these data exchange, we need another model: the distributed-memory model.

2.3.2 Distributed-memory models

With distributed memory, each processor has its own private memory. Tasks running on a processor or node can only operate on data from this processor or node. If data from another processor or node are required, tasks must communicate with each other. Usual distributed memory systems are composed of nodes with their own memory and an interconnection mechanism that allows data exchanges between the nodes. Each system builds the interconnection system as it wishes. There are multiple common ways, like Ethernet for example. The key with distributed memory is to determine the best way to divide the application into small parts that can work on different data items. There exist libraries and standards created to abstract data exchanges and their complexity to the programmer. The most common and widely used in HPC is MPI, the Message Passing Interface [16].

2.3.2.1 Message Passing Interface (MPI)

MPI is a portable message-passing standard designed to operate on a wide variety of parallel computing architectures. The standard only defines the syntax and semantics of functions useful for application developers writing message-passing programs in C/C++ and Fortran. Thus, their exists multiple implementations of the standard in libraries, some open software like OpenMPI [17], MPICH [18] for the most known or proprietary like Intel-MPI (derived from MPICH). The core of MPI is a number of functions used to send messages between processes (called MPI tasks, or workers).

A program using MPI will launch multiple workers on a machine. Any number can be launched anywhere on the machine. For example, it is possible to launch multiple workers on only one processor. However, for larger simulations, multiple nodes can be used. Each worker possesses its own memory, and messages can be sent between workers with MPI-defined functions using what MPI calls communicators. Communicators are a set of workers that can send or receive messages from each other. Workers can be in any number of communicators. At the start of the application, MPI creates the World communicator containing all the workers, but the user can create its own as he sees fit.

The benefits of MPI are mainly that it is possible to create as many workers as one wishes, thus making possible for simulations to use a very large number of workers on huge machines. Moreover, MPI is not architecture dependent. If a machine is built with an interconnect between processors and nodes, MPI can be used and applications can be ported to it. On the other hand, MPI workers are usually processes. Creating multiple processes on the same processor is not optimal as it wastes memory [19]. Moreover, sending messages between workers

that share memory with each other is inefficient. Indeed, why go through a message sending protocol and waste time when shared memory could be accessed all participants?

To remedy to this problem, some MPI implementations are starting to use threads. Other models where created from the start to implement workers as threads (MPC [20]).

2.3.2.2 Distributed-memory model conclusion

Distributed memory models are important in HPC as they allow user to launch large simulations on clusters of multiple nodes and processors. However, they can be less efficient at using shared memory than dedicated shared-memory models. Fortunately the two models are not exclusive. Simulation codes can use both distributed and shared memory models to design powerful application that can run on large clusters.

2.3.2.3 Hybrid programming

To combine the best of both shared and distributed memory, simulation codes are starting to use both models at the same time. Hybrid programs creation seems natural when looking at clusters architectures. For example, a distributed-memory model could be used to create one process per node, and manage communications between nodes, and a shared-memory model would populate nodes with threads to benefit from the shared memory of nodes. Multiple runtimes implementing these models exist. And these implementations can be largely different.

Let us take OpenMP classic usage and OpenMP tasks (both seek to exploit shared memory). However, their usage is very different. While the first is based on the fork-join model, the second creates tasks scheduled independently. The example of OpenMP shows multiple approaches in the same runtime, but to a certain extent each runtime has its own specificities. Some runtimes are even specifically created for certain architecture. We can think of Cuda for NVIDIA GPU usage for example. Using multiple runtimes in a code increases its complexity for the development phase and later optimizations. When creating an application, user should choose which runtimes to use carefully.

With the increasing complexity of architectures, using multiple runtimes at the same time is becoming mandatory to exploit the maximum performance. This leads to situation where multiple runtimes are running concurrently during execution. We call these situations **runtime stacking**.

2.4 Conclusion

As we saw in this chapter computer architecture has greatly evolved in the past years. Hardware improvements are somewhat at a standstill but improvement are still made. Breakthrough both in hardware and software are being or will be made to reach the exascale milestone.

On the hardware side, one of the major trend of the past decade was the increase of the number of cores inside processors, either regular CPUs or dedicated resource units. This leads to the rise of multicore technology with irregular accesses (NUMA nodes, cache rings or meshes...) and the advent of manycore architectures like NVIDIA GPGPUs and Intel Xeon Phi. Even if other innovations are currently on tracks for hardware development, providing an increasing number of compute units per chip is still a major evolution axis for next generations of HPC (High-Performance Computing) supercomputers.

On the software side, and more precisely regarding the parallel programming models available to exploit those compute units, MPI is widely used by most of parallel applications. However, recent studies show that scalability issues will show up at a large scale [19]. Since manycore processors exhibit shared-memory properties among numerous cores, a natural idea to exploit these hierarchical architectures is to use *threads* to match the memory-sharing capabilities of the hardware. Therefore, one typical direction is to mix MPI with a thread-based model exploiting the shared-memory system leading to MPI+X programming.

Chapter 3

Problem

With the advent of multicore and manycore processors in HPC clusters, many applications are mixing distributed and shared-memory models. Mixing models involves various runtime libraries to be alive at the same time and to share the underlying computing resources. As clusters architectures become more and more complex, more heterogeneous with specialized components, the number of threads and runtime is also likely to grow. Thread placement has become an important optimization focus. Threads engaged in a lot of inter-node communications will need to be placed near the network card, threads in charge of I/O will need to be placed near the memory and so on. All these threads and runtime sharing resources also means that efficient resource utilization will become critical. An ineffective resource usage could overload some resources while keeping others idle thus impairing performance to a large degree.

However, little research has been conducted on runtime stacking. The major issue is that resource usage is not noticeable at execution time. The operating system is not easily relinquishing information about what is happening to users. A basic user won't even imagine that threads could misbehave. And he would be going out of its way by parsing kernel information to determine thread placement and resource usage.

In the end, there is no easy way to look at resource usage. No tool to observe how resources are used. No error at execution time when misuse happen or at compilation if misuse could happen. Nothing to help optimize placement and resource usage.

Moreover, while mixing two models together may improve the application performance, it adds a new level of complexity for the code development. Indeed, runtime libraries implementing those models are not usually designed to be inter-operable with each other. It would be illusory to imagine all runtimes taking each other into account. Thus, most of the time, they ignore

each other and it is the user's duty to ensure that everything works together. Threads/processes created by different libraries are scheduled onto hardware resources by the system scheduler, most of the time without any knowledge about other existing execution flows, leading to potential cache interference, synchronization overhead or unnecessary time loss in communications. Furthermore, HPC applications like simulation codes often rely on calls to optimized libraries or different solvers to reach high performance. But each library may be parallelized with different models. For example, an MPI application could deploy a solver based on OpenMP (controlling the mix of two runtime libraries: MPI and OpenMP) and then call a second solver parallelized with Intel TBB within the same time step. This would lead to deal with 3 models at the same time.

On the user's side, they can mostly only interact during the allocation phase. They can have an influence on the batch manager and ask for the right number of nodes and tasks, sometime the number of threads for certain runtimes. Then the runtimes are running amok on compute resources. Thread based models in particular can create any number of threads, not knowing if other models are concurrently accessing the resources.

Then what are the options left to the user? Use default parameters and hope for the best or try to optimize parameters, but then there is no easy way to observe what is going on. Only application's performances will tell if some parameters are better or worse than others, but it will not determine if a set of parameter used resources efficiently. The next best thing is to look at algorithms and do scalability studies on the codes. And when all of this is done, and the code is ported to a new cluster, optimization might not be portable.

This thesis is an effort to study runtime stacking and develop a shared interface to manage resources when multiple runtimes are sharing resources.

3.1 Motivating Example

Porting an application to various hardware/software environments can be challenging: the same execution configurations can provide different performance results. To illustrate the necessity to focus on runtime interactions et resource usage we present the following motivating example. We tested the execution of the CORAL Benchmark Lulesh [21][22] on two Intel processors: a Xeon Haswell (16 cores) and a Xeon Phi Knight's Landing (KNL with 64 cores). Both architectures support hyperthreading (Intel SMT): 2 hyperthreads per core on Haswell and 4 on KNL.

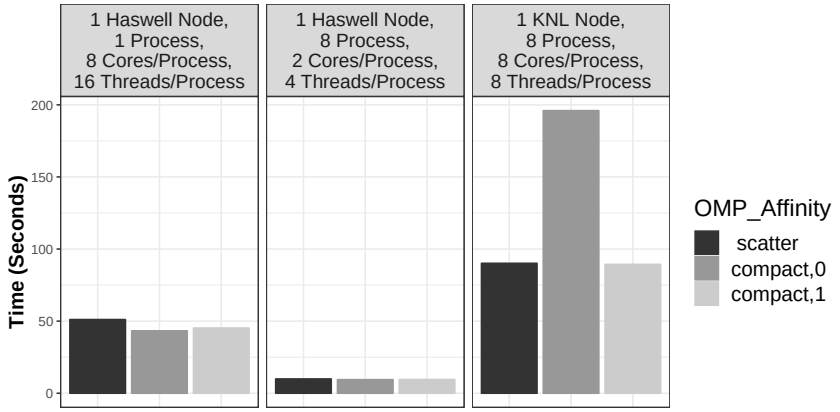


Figure 3.1.1 – Execution time running Lulesh benchmark on multiple configurations.

Figure 3.1.1 depicts the execution time running Lulesh with various configurations. The first set of bars represents the elapsed time of 3 runs on 1 Haswell node with 1 MPI process and 16 OpenMP threads on 8 cores. This configurations is exploiting one socket and all its hardware threads. The second set represents the elapsed time of 3 runs on 1 Haswell node with 8 MPI ranks and 4 OpenMP threads per process. This time, each process is using two cores, thus exploiting the entire node. The last set is for 3 runs on 1 KNL node also with 8 MPI ranks on 8 cores and 8 OpenMP threads per process. The black bar is used for the *scatter* placement policy of OpenMP threads, the dark grey for the *compact,0* policy and the light grey for the *compact,1* one.

For the first configurations, 16 threads are spanned across 8 cores, exploiting 2 hyperthreads per core. The *compact,0* policy is the best because it gathers the threads close to each other, considering hyperthreads. For example, OpenMP threads with rank 0 and 1 will be on 2 hyperthreads of the same core. The *compact,1* gathers the threads close to each other but considering cores, not hyperthreads. For example, OpenMP threads with rank 0 and 1 will be on 2 different cores. Once the cores are populated, we start binding the threads to the other hyperthread following the same logic. If the data locality is better than the *scatter* policy, it is still a bit worse than *compact,0*. However, as all the threads and data are on the same socket, the performance difference is small, but noticeable.
The second configuration using 8 processes still exploits all hyperthreads of the Haswell node but now also all the cores of both sockets. The multiple MPI processes reduce the overall execution time. They also limit the impact of data locality as less thread per process are created. Hence, the performance are almost identical for the 3 thread placement policies.

However, the last configuration on a KNL node shows different results. Here, the *compact,0* policy provides the worst performance, contrary to the previous two examples. With this configuration, threads are gathered on the hyperthreads of the first cores of the KNL. Hence, instead of using 64 cores, the 64 threads are folded on 16 cores. All the hardware threads of these 16 cores are exploited while the 48 remaining cores stay idle. In this case, using the hyperthreads reduces performance compared to both *scatter* and *compact,1* policies which spanned the 64 OpenMP threads on 64 different cores.

3.2 Contributions

This simple example shows that the same configuration is not portable across platforms. It also shows the impact of thread placement when mixing worker threads from multiple libraries. However worker threads are not always the only ones exploiting computing resources. The system has threads running to perform various tasks for example. Some runtimes can also use helper threads. The use of progress threads to perform non-blocking communications in MPI [23] is an example. Studies showed that their use and placement can have a big impact on communication performances [24]. To provide the best configuration, a lot of parameters should be taken into account. To tackle this issue, this thesis makes the following contributions:

- Chapter 4, introduces runtime stacking *configurations* and *categories*. The categories are a classification of the methods used to mix multiple runtime in a code, while the configurations explore what this mixing translates to at execution time.
- Then Chapter 5 presents algorithms designed to check for resource usage issues. It also presents a tool implemented during the thesis called *the Overseer*. This tool produces logs of resource usage by runtimes at execution time. The logs are then studied with the helps of the algorithms, to determine if any resource misuse happened during the execution of the application.
- Finally, Chapter 6 presents a second tool called *the Overmind*. It catches every worker creation and, with the help of the algorithms, dispatches them on available computing resources to avoid their misuse.

Part II

Contributions

Chapter 4

Taxonomy

4.1 Taxonomy: introduction

On the hardware side, one of the major trend of the past decade was the increase of the number of cores inside processors, either regular CPUs or dedicated resource units. This leads to the rise of multicore technology with irregular accesses (NUMA nodes, cache rings or meshes...) and the advent of manycore architectures like NVIDIA GPGPUs and Intel Xeon Phi. Even if other innovations are currently on tracks for hardware development, providing an increasing number of compute units per chip is still a major evolution axis for next generations of HPC supercomputers.

On the software side, and more precisely regarding the parallel programming models available to exploit those compute units, MPI is widely used by most of parallel applications. However, recent studies show that scalability issues will show up at a large scale [19]. Since manycore processors exhibit shared-memory properties among numerous cores, a natural idea to exploit these hierarchical architectures is to use *threads* to match the memory-sharing capabilities of the hardware. Therefore, one typical direction is to mix MPI with a thread-based model exploiting the shared-memory system leading to MPI+X programming. While mixing two models together may improve the application performance, it adds a new level of complexity for the code development. Indeed, runtime libraries implementing those models are not usually designed to be inter-operable with each other. Threads/processes created by different libraries are scheduled onto hardware resources by the system scheduler, most of the time without any knowledge about other existing execution flows, leading to potential cache interference, synchronization overhead or unnecessary time loss in communications. Furthermore, HPC applications like simulation codes often rely on calls to optimized libraries or different solvers to reach high performance. But each library may be parallelized with different models. For example, an MPI application could deploy a solver based on

OpenMP (controlling the mix of two runtime libraries: MPI and OpenMP) and then call a second solver parallelized with Intel TBB within the same timestep. This would lead to deal with 3 models at the same time. Even if each parallel programming model may be relevant for some pieces of code, mixing them creates a *runtime-stacking* context that may lead to large overhead if the configuration of each library and the global environment are not properly set.

This chapter presents a study of runtime stacking. It introduces stacking configurations and categories to describe how stacking can appear in applications. More specifically, we will first present an identification of runtime-stacking categories to illustrate the different situations where mixing multiple parallel programming models may appear (explicitly managed by the end-user or not). In a second part, we will explore runtime-stacking configurations (spatial and temporal) focusing on thread/process placement on hardware resources from different runtime libraries. The focus is on resource usage, and more specifically core usage. This work could be extended to any resource shared by runtimes. Memory is the first shared element that comes to mind, but with the rise of more specialized, heterogeneous architecture, we can imagine having to share I/O devices, co-processors and accelerators and so on.

This work as been published in 2017 in the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) [25].

4.2 Stacking Configurations

This section presents the study of different runtime-stacking configurations that may appear when a hybrid application (with multiple programming models) runs on a cluster. Indeed, at execution time, the different threads / processes are scheduled on computational resources and the runtime libraries (corresponding to programming models) are scattered and executed across the machine. Depending on many parameters (including the machine configuration, scheduler policy, runtime implementation, resource reservation method, hints, environment variables...), the eventual placement of threads from the whole application may vary. Based on this placement and resource usage, we define the notion of *runtime-stacking configurations* to represent how execution flows (threads and processes) are scheduled on a target machine. The main goal of these configurations is to define and understand how the underlying runtime libraries interact with each other in the application. We can then investigate on the effects these interactions can have on performances.

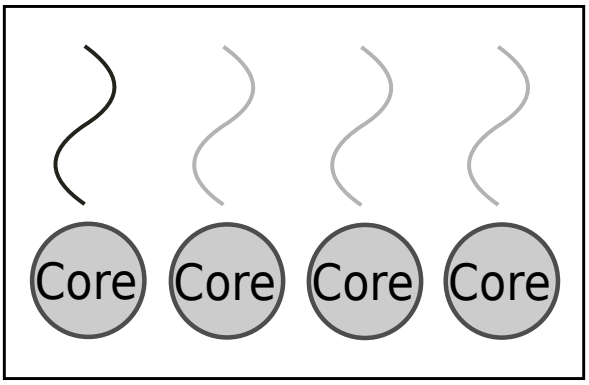


Figure 4.2.1 – Runtime Stacking Configuration: Spatial Independent. The black and grey runtimes are using different cores.

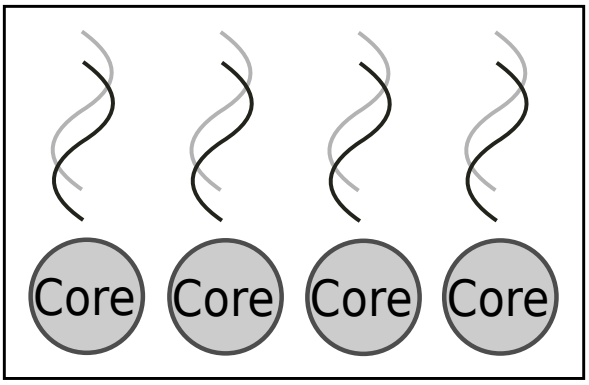


Figure 4.2.2 – Runtime Stacking Configuration: Spatial Concurrent. The black and grey runtimes are both deployed on all the cores.

4.2.1 Spatial analysis

When looking at interactions that could happen between two runtimes, we can first look at which resources these runtimes are running on. More specifically, we want to determine if they are sharing resources or not. We call this observation `Spatial Configuration` of runtimes.

Runtimes sharing hardware resources may involve compute cycle stealing and therefore performance loss. Moreover, switching from a thread to another or worse from a process to another is also time-consuming. When switching threads, only processor state (program counter and registers content) has to change. The virtual memory space remains the same. However, the harmful effect is more felt in the caches. If threads are not working on the same data, chances are that accesses will produce cache misses, wasting cpu cycles. These effects may deteriorate performances of the applications. Thread switching should be avoided if possible, but is all in all generally efficient. Context

switching between processes on the other hand is a different kettle of fish. Process context switching involves switching the memory address space. This includes memory addresses, mappings, page tables, and kernel resources. On some architectures, it even means flushing various processor caches that aren't shareable across address spaces, this can go from the TLB to entire L1 cache.

Two runtime libraries can be *spatially independent* if they deploy execution flows on disjoint resources. More specifically, each flow is scheduled on separate resources and each compute unit is busy with one execution flow. In this case, there is basically no interaction between the two available runtime libraries at the hardware level because the set of resources is disjoint. Moreover, as there is only one flow to execute per compute unit, there is less risk of context switching between processes or threads, thus keeping good performances with caches and other shared hardware resources. Figure 4.2.1 illustrates this configuration. Two runtimes, one black using one thread and the second grey using three threads are running on a four core node. Each thread is using its own core, they are all using independent sets of resources. This configuration is often the first target configuration when developing and optimizing a hybrid application exhibiting runtime stacking.

On the other hand, if some execution flow competes for the same resource, we enter the configuration called *spatially concurrent*. This configuration may involve compute-cycle stealing, cache thrashing and other effects harmful for applications' performances. Figure 4.2.2 illustrates this configuration. This time the black and the grey runtime both create four threads on the four core node. Each runtime places its threads on a different core. In the end each core is used by one thread of each runtime, and both runtimes are using concurrent sets of resources.

In this context, one key parameter can be the *wait policy* which drives the way each programming model will put the corresponding thread asleep when not active. For example, in OpenMP, it is possible to choose between *active* and *passive* mode [26]. With active mode, each thread waiting for work will consume CPU cycles. In other words, they still monopolize resources for some amount of time when they are in a waiting state. It results in better reactivity when starting a new parallel region as the same thread can be used again without context switches. On the other side it may reduce the performance of the other model performing computation on the same cores at the same time, as they need to wait longer to acquire resources. On the other hand, passive mode allows better interoperability but may lead to poor performance because threads may take more time to wake up for the next parallel region. Experimentations on micro-benchmarks showed what the overhead of entering and exiting parallel regions could be up to 10 times larger in passive mode [27].

There are different situations where this configuration may appear. First, users may ask as many threads as the number of available cores on the target compute nodes (with environment variables like `OMP_NUM_THREADS` or some parameters of job manager). Another scenario is a bad resource allocation for the different runtime libraries. Indeed, if all programming models are not aware from each other, they may want to deploy their execution flow on the whole node.

This spatial analysis is important as many optimizations possibilities can arise from it. Let's take the example of an iterative simulation. For each time slice, two solvers from different libraries are called. These two solvers are working on different set of data and synchronising at the end of each iteration. The target architecture is composed of multithreaded cores. For the sake of simplicity, in this example we will assume users want to use all the available resources (all hardware threads), and divide them equally for each runtime. As solvers and runtimes are not aware of each other, the most likely scenario is that when reading programs arguments each runtime will create a number of threads equal to the number of available core and place one thread per core. In the end, each core will run one thread of each runtime. As each solver is working on his own data set, this will lead to cache thrashing situation described above. Another solution would have been to place all threads from the first solver on the first half of the node using all hardware threads of cores, and the threads from the second solver on the other half. This way the chances of cache thrashing are less likely. Moreover, if the node was a NUMA node, this configuration would decrease or negate NUMA effects [28].

In the end, being aware of runtimes placement and optimizing resource usage is important. But there is another factor to take into account: thread scheduling and runtimes' lifespan. Next section describe our temporal analysis of runtime behaviour.

4.2.2 Temporal analysis

When looking at interactions that could happen between two runtimes, we also need to look at runtimes' lifespan. We call this observation `Temporal Configuration` of runtimes.

We define two programming models as `temporally independent` if there is no overlap in their creation-destruction time frame. Figure 4.2.3 illustrates this principle. We represent two runtime lifespan on a timeline, the first runtime in black, the second in grey. On the figure, there is no overlap on the lifespan of the runtimes, they are temporally independent. If two models are temporally independent (they are never alive at the same time) whatever the resources they are on, they will never be spatially concurrent. Moreover, as they are not

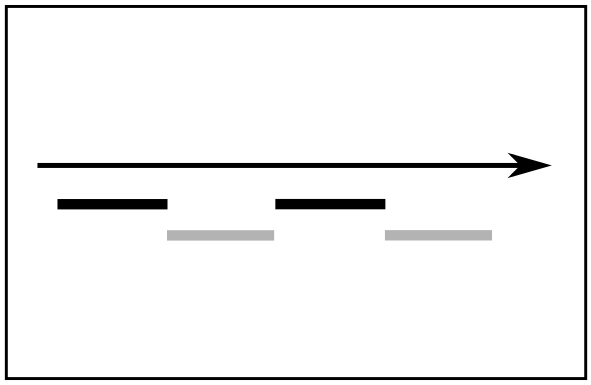


Figure 4.2.3 – Runtime Stacking Configuration: Temporal Independent. The black and grey runtimes are never running at the same time.

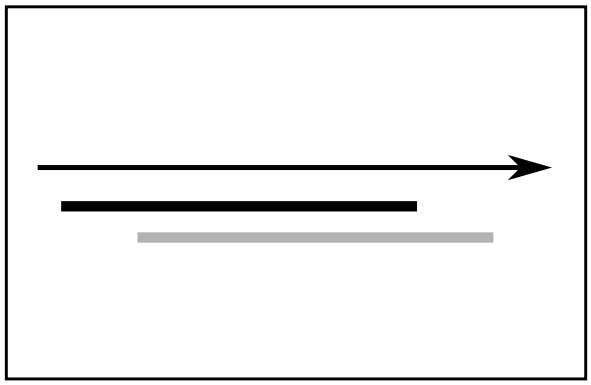


Figure 4.2.4 – Runtime Stacking Configuration: Temporal Concurrent. The black and grey are both running at the same time during a time period.

alive at the same time, there is no message transiting from one to another, so no possible NUMA effect or waiting for messages/answer.

This situation arises for example when a code performs multiple repetitive calls to an optimized parallel library. Each call will spawn and deploy library-related threads, but each instance of these runtimes will be independent of each other. Conversely, two runtimes are said to be `temporally concurrent` if they can be scheduled in the same time frame. This is the case when an MPI+OpenMP hybrid code makes MPI calls from within OpenMP parallel sections for example. Figure 4.2.4 illustrates this principle. Here the black and grey runtimes are alive concurrently during a certain period of time.

While temporal independent runtimes don't cause any performance issue, temporal concurrent ones can. First, if two runtimes are temporal and spatial concurrent the probability of issues described in the precedent section arising is very high. But even if runtimes are not spatial concurrent, there is still

space for performance degradation and optimization opportunities. Indeed, two runtimes communicating with each other on the same node can be susceptible to NUMA effect. Thus placing runtimes communicating with each other the closest possible can be an effective strategy.

Runtime threads placement is unfortunately not easy to optimize. First there are a lot of factors to take into account: architecture used, its number of cores, socket configuration, memory placement, but also the number of runtimes, their time frame, communication intra runtimes but also communication between runtimes and so on. The next section discuss some of these concerns.

4.2.3 Discussion

The definition of runtime stacking configuration is a first step to describe what could happen when mixing runtimes. It also pinpoints possible runtimes interactions at execution time. However, these clean configurations are almost always not applicable to whole applications. During the entire execution of an application, multiple `runtime stacking configurations` may alternate. In fact, these configurations may also vary from execution to execution depending on multiple factors, with some of them, like scheduling, not being entirely predictable.

By looking at how execution flows were scheduled, we could determine if there is a contention issue with parallel models, or if every instance was scheduled on its own set of resources without interference. Knowing which configuration appears is also valuable information when developing and optimizing a code, as spatially independent runtimes is often what programmers are looking for.

Another information to take into account when trying to understand how runtimes can interact together is the way they are used by application developers. The next section describes what we call `runtime stacking categories` which describe and sort techniques used to mix parallel runtimes in codes. These stacking categories coupled with the stacking configurations describe how runtime stacking is created and what happens at execution. All the information gathered are the basis to understand how to optimize runtime placement.

4.3 Stacking Categories

As mentioned previously in Section 2.3, relying exclusively on MPI programming model may lead to scalability issues, which explains why MPI is increasingly mixed with thread-based models to improve overall performance.

But this is only one example where model mixing appears. Indeed, calling external libraries or relying on other models (more abstracted) may involve runtime stacking situations. In addition to multiple runtimes being mixed, one runtime can create multiple instances or worker groups that could interfere with each other. This is the case with nested OpenMP parallel section for example. Some runtime can also make use of helper threads, created at execution time without the intervention of the user. This can be the case when using progression threads with non-blocking collectives communications in MPI [23]. To explore those situations, we introduce *Runtime Stacking Categories*. We've described how runtimes can interact with each other at execution time. This section now describes how these interactions were created in the first place and defines how to mix runtimes together in an application's code.

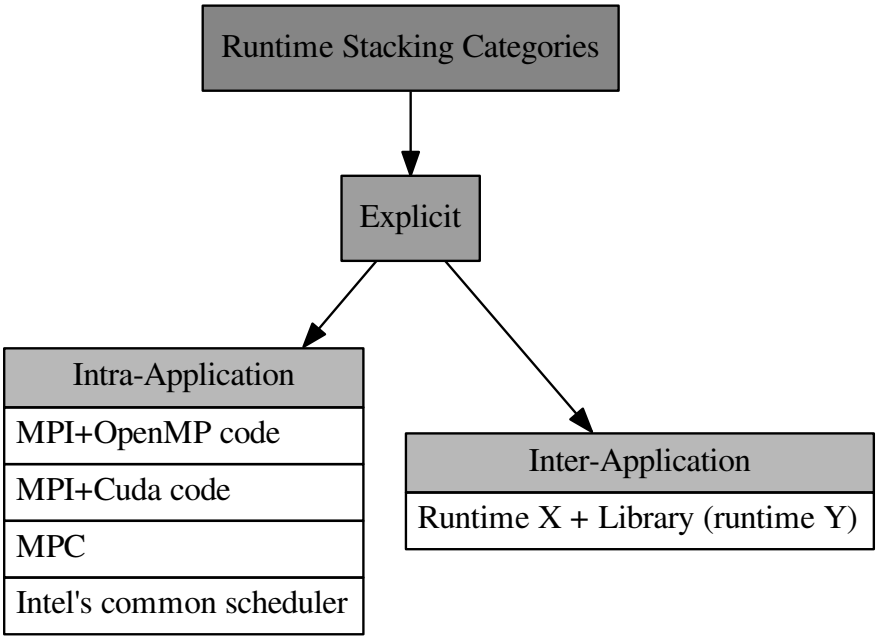
4.3.1 Explicit addition of runtimes

The most logical thing to do when we want to add threads or process to a code is to make function call that create threads or processes. MPI, OpenMP or Pthread are examples that use functions calls to create and manipulate threads and processes. When using multiples of these runtimes in a code, their stacking is explicitly exposed. This is the most common way to lead to runtime-stacking context in a code. We call this approach the *explicit* approach. It means that application developer might be aware of this stacking situation and can take actions to change the configuration and improve performance.

4.3.1.1 Intra-application

Inside this *explicit* large category, we define two sub-categories 4.3.1. The first one, *intra-application* stacking groups the applications that explicitly call different runtime libraries, for example in a hybrid MPI+OpenMP code. Thus stacking is directly exposed by the programmer through MPI function calls and OpenMP directives. It is actually the solution that offers the best flexibility in thread creation, number, behaviour and so on. Yet, even if relying on various optimized parallel runtime libraries would seem natural, most applications only use one or two of them at a time. Indeed, mixing more than a couple of runtimes takes a lot of knowledge. First the application's code. This includes the algorithms used but also all the runtimes used. In particular how and when they are alive, what kind of interaction they can share and so on. Then knowledge of the architecture can be extremely useful to optimize resource management, number of threads, or the algorithms themselves. Finally, other considerations can be explored like scheduling algorithms, runtime parameters, and so on. Even though this method has the potential for the best optimiza-

Figure 4.3.1 – Runtime Stacking Categories: Explicit



tions, it is also error prone. First because of the large number of parameters to fine tune. But also because models were not designed to run concurrently on the same resources. This problem is known as the composability problem [29].

There exist initiatives that help users writing optimized hybrid code. For example, MPC [20], [30] is a framework that provides a unified parallel runtime designed to improve the scalability and performance of applications running on HPC clusters. It allows mixed-mode programming models and efficient interaction with the software stack by providing its own MPI, OpenMP and Pthread implementations, based on an optimized user-level scheduler. However, it does not give the user any feedback about resource usage. With the same objective of composing models, Intel has been using a shared runtime system basis and scheduler [31], [32], to limit the interference between their runtimes. If used in the same application, Intel TBB and Intel Cilk can run concurrently by sharing the underlying task scheduler, and thus avoid thread over-subscription.

4.3.1.2 Inter-application

The second sub-category in the *explicit* group is *inter-application* stacking. This category encapsulated techniques that exhibit stacking through calls to external libraries. For example, writing an MPI code and calling libraries based on OpenMP (e.g., BLAS MKL) or Intel TBB is fairly common. In this case stacking is indirect, as calls to different parallel models are made in external libraries. Here the programmer still gets to manage algorithms, and messages between nodes but some runtime intricacies are delegated to libraries. The need to understand the runtime used in the libraries is less important than with *intra-application* stacking. In fact, knowing how, when, or why the second runtime is used might not be necessary at all to create a sound, working application. However, by knowing the library implementation (or at least the parallel programming model they rely on), stacking can still be explicitly observed and be influenced.

In the end, even though the sources of the libraries are most likely not accessible to the user, use of runtimes is still explicit. All calls to them are made by the user (even indirectly by using a library). Optimization in algorithm, runtime calls, thread number and behaviour still might be controllable although to a lesser extent as with *intra-application* techniques.

Note that libraries creating and managing processes and/or threads often try to improve performance by bypassing the system scheduler through direct allocation and binding of execution flows. As each library may be unaware of other resource utilization, the target code will potentially exhibit poor performance due to those libraries interferences. Moreover, influencing these libraries' threads behaviour might be limited to features brought up by libraries.

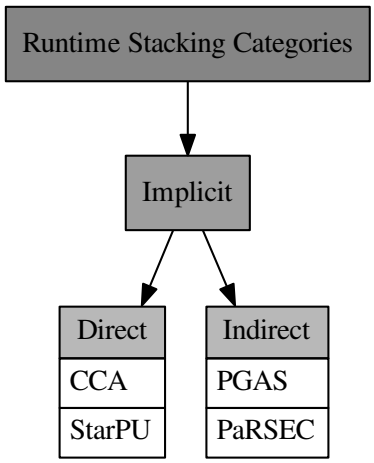
4.3.2 Implicit use of runtimes

The *explicit* category encapsulates every runtime stacking technique that make application developers explicitly manipulate runtimes. The explicit call of functions and libraries make the use of specific runtime more or less obvious. The next section presents techniques that make the use of runtimes transparent to user. This category is called *implicit* runtime stacking. It represents approaches where end-users might not know which runtimes will be used at execution time. In this case optimization possibilities might be limited.

4.3.2.1 Direct addition

This category is also subdivided in two sub-categories 4.3.2. The first one is called *implicit direct*. It represents cases where programmers rely on unified or abstracted platforms designed to help them compose pieces of codes.

Figure 4.3.2 – Runtime Stacking Categories: Implicit



The Common Component Architecture (CCA) [33] is an example of implicit direct stacking. Components are black boxes used to build an application. Each component can be parallelized directly with regular models (e.g., MPI, OpenMP or NIVIDA CUDA) but when using them, we may or may not know how they were optimized and which runtimes were used. StarPU is another example of platform that helps users stack models [34]. The runtime-system goal is to manage parallel tasks over heterogeneous hardware. StarPU relies on a hypervisor to dynamically chose which implementation of a kernel will be more suitable for the target hardware resources. Moreover, it uses a dynamic resource allocation with *scheduling contexts* [35]. The use of a global hypervisor may mitigate scheduling problems, however tasks are still coded with classic runtimes, thus creating situations where runtime stacking issues may appear.

With this category, knowledge of runtimes by programmer is here not needed at all. The underlying platform will take responsibility for scheduling, creation and behaviour of runtimes. However, the more common runtimes (MPI, OpenMP, TBB, Cuda, ...) are still used, and a lot of parameters are to take into account. Familiar issues like allocation sizes might still arise. Optimization opportunities are still present when dealing with runtime parameters. So, even though knowledge of the runtimes is not mandatory, a basic (or advanced) knowledge of runtimes, architectures, and more, is still a huge advantage when optimizing applications.

This category gets its 'direct' qualifier from the fact that runtimes used in the code are still the common ones. They are still used in a 'direct' manner as

they would in codes from the *explicit* category. This means that the person programming a part of the code using runtimes is aware of there presence. However, at execution time, the model will chose which part of the code to execute and when, thus the stacking remains implicit. This is in contrast to the next sub-category where code introduces new indirect techniques to use programming models.

4.3.2.2 Indirect addition

The last sub-category called *implicit indirect stacking* represents approaches where end-users may have difficulties knowing what happens at execution time. In particular, if multiple parallel programming models are actually used at the same time. This category includes PGAS languages that abstract the way to communicate and share memory, relying on different models to improve performance. For example, a PGAS library implementation may rely on MPI for inter-node communication and regular threads for intra-node synchronizations. On the other hand, the library could also implement its own methods to use and share intra-node memory and send messages between nodes. It could just as well use all of these methods depending on the situation. In these situations, the implementation is in charge of selecting the best combination and helps the application end-user to configure the resource usage on supercomputers. This category also includes new approaches like PaRSEC [36], an event-driven runtime that handles task scheduling and data exchanges which are not explicitly coded by the developers.

Here the end user only needs to know how to use the specific language. Knowledge of runtimes used is not necessary. Often, knowledge of underlying architecture is not either. Runtime optimizations outside built in language functionalities are usually not possible. With a deep knowledge of the specific library and inner functioning optimizations should be feasible but it does not seem reasonable as it is not the focus of these implementations.

4.3.3 Discussion

The categories highlight the fact that there are many situations that may exhibit runtime stacking. Each of the approaches described in this section is relevant and has its own advantages/drawbacks depending on the target hardware resources and the current state of the parallel application. The common aspect is that eventually, multiple model implementations may coexist during the execution of the application leading to poor resource usage if parameters are not correctly set. Unfortunately the multitude of models and techniques to implement them make easy cross platform optimization solution impossible.

One solution would be to have every runtime aware of each other. This way, they could share resources without interfering with each other unnecessarily. They could even 'discuss' and exchange resources when needed. However, this solution seems utopian as there are already a lot of existing models. And a lot a more models will certainly be implemented in the future. Implementing knowledge of multiple other existing and yet to exist models is not feasible. On the other hand, this is a huge stepping stone for future improvements. Indeed, revise all existing runtimes is not achievable, making a couple of the most common ones run together is feasible. Besides, as we have seen earlier in 4.3.1.1 initiatives that couple runtimes already exist (MPC and Intel's shared scheduler).

However, if we look at all these runtimes we can see a common attribute. Indeed, the only way to create parallelism on a computer is by the use of processes and threads. These processes and threads are the one exploiting the computer resources in the end. This is the entry point that could help optimize resource usage regardless of the runtime, model, or architecture, or any other parameter used.

This thesis is a look at what it is possible to achieve when manipulating threads, with the knowledge of runtimes and stacking issues. By working at the thread level, we could get information about the models and implement additions usable by all categories. However, as explicit stacking is the most represented category in HPC at the moment, we focused on applications using these techniques. Moreover, usual runtimes like MPI or OpenMP are more widely known and accepted than domain specific languages. They are widely used and easier to study and get information about. Our work is still generic though. We could get the same information from any runtime.

4.4 Taxonomy conclusion

Figure 4.4.1 – Runtime Stacking Configurations

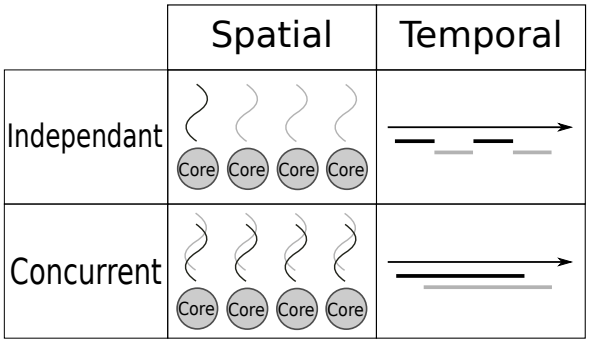
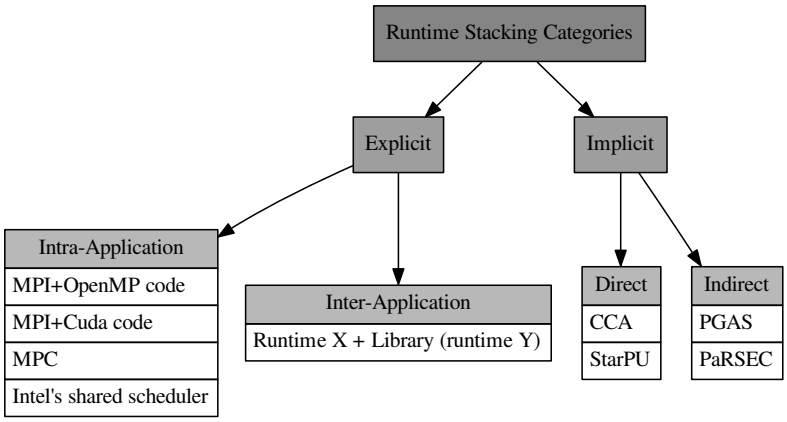


Figure 4.4.2 – Runtime Stacking Categories



The two principles presented in this taxonomy describe runtime stacking. The configurations 4.4.1 illustrate how runtime share resources at execution time. If runtimes are temporally concurrent, they can interact with each other whether it is by sharing resources (in which case they would also be spatial concurrent) or by sharing data via messages or shared memory. Categories 4.4.2, on the other hand, describe how runtime stacking code are written. More especially the techniques that create codes using multiple parallel libraries.

However, explaining these principles is not enough to solve the issues caused by runtime stacking. Indeed, there is no convenient technique that would point runtime stacking issues to a user.

Mixing multiple runtimes adds a layer of complexity to already error prone parallel codes. However, there is no mechanism to help debug, or even point to the problematic code segment. There won't be any warning at compilation time, neither error message at execution time. Everything will work fine, if we pass over the bad performances. And even if we detect less than optimal performances, we don't have any clue about the cause of the problem. Is it an algorithm problem? Is it from resource usage? Is it from resource allocation? Is it something else?

Using the principles from the taxonomy, the next chapter presents the algorithms designed to detect resource misuse at execution time.

Chapter 5

Algorithms and Tools

The last chapter introduced runtimes stacking `categories` and `configurations`. It also pointed out the lack of tools to observe the state of runtimes' resource usage. The motivating example in Chapter 3 also pointed out the tremendous effect that bad runtime parameters could have on application performances. It also showed less dramatic experiments where a small mistake would also lead to resource misuse and lost performances.

The main problem with these resource missuses are hard to detect. The only possibility would be to notice an unusually long execution time. However, where a huge overhead can *maybe* be detected, a smaller one will most likely not be. In both cases nothing except execution duration observation and scalability studies would give hints that a problem is present. Then finding the error's cause could take a long time. First the problem would have to be narrowed down to resource usage which is not an easy task. Multiple others issues could cause the same performance degradation. Is it a problem in the algorithm? In its implementation? Somewhere else? And finally, when the resource usage issue is found, there are still many factors to take into account. Resource allocation, number of processes and threads, processes and threads placement, scheduling and so on.

This section proposes algorithms to check for all resource usage issues. The idea is to compare resources available to threads and processes using them. This way, we know at every step of the execution which number of resources are used, and we can check if it is what we actually planned to use. We also can see all processes and threads using them and determine if resources are used according to our plans. In the end, if a misuse is exposed, optimising or debugging is easier. We already know if the problem came from the resource allocation, and which runtime created what number of threads and on which resources.

5.1 Bibliography

Numerous tools exist for debugging and optimization, but none of them consider runtime stacking issues. Indeed, our goal is to determine if threads spawned by different runtimes (or by different instances of the same runtime) are competing for resources. Therefore it is necessary to track thread/process placement looking for their spawn site and determine if there is a race for common resources.

In this context, debuggers like GDB [37], its parallel counterpart DDT [38], and others like Totalview can track each thread position, sometimes along with which runtime spawned them and if they were busy or not. However, users need to manually track each thread, to retrieve significant information, and to perform the analyses by hand to look for resource conflicts. Moreover using a debugger can alter the dynamic behavior of threads and may actually prevent the detection of such problems. Furthermore debuggers working on runtime error detection, like Marmot [39] or MUST [40], will no help either. Runtime stacking only influence the application performance without raising any error, and will by transparent to such tools. In the end, using a debugger to detect stacking issues may not be the right tool.

Since runtime stacking has an impact on performance, performance analysis and profiling tools such as Scalasca [41], TAU [42], HPCToolkit [43] or SCORE-P [44] may detect the issues. These tools can pinpoint the scalability bottlenecks as communications and synchronizations. However the only resource related analysis is per-rank load balancing. Visualization tools like VAMPIR [45] or Paraver [46] may be able to expose thread placement and usage but once again stacking is not taken into account.

Multiple tools can be combined to effectively detect problems due to runtime stacking. But this effort can be large because of code modifications (for specific instrumentation) or high dynamic overhead. This modification of behavior may change the dynamic stacking. Therefore, there is a need for light-weight tools which would help the end-user to monitor the application execution and extract the resource usage, this information then being used to determine stacking-related errors or misuses.

5.2 Algorithms detecting resource usage

To obtain the big picture of what happened at execution time, and help optimising flow placement as well as resource usage, we present algorithms which take as input execution information of an application (through traces or online events) and determine the corresponding resource usage. They detect

warnings if the load of the machine is detected as non-optimal (overloaded resources or idle resources).

We present two main algorithms. First, we detect wrong usage of resources through execution flows, and then we focus on each resource to check if each of them is busy or not. In this thesis, we focused on threads (flows) and cores (resources) but note that these algorithms could be used to detect misuses of other hardware resources (compute units, memory, IO components) from different flows (instructions, data, messages).

Algorithm 1: Flow-Centric Algorithm

Data: $Flow = F_0, ..., F_n$, $Resource = R_0, ..., R_n$

$S \leftarrow \emptyset$
foreach $F \in Flow$ **do**
  $S \leftarrow S \cup R_F$
end
if $(\sum_{i=0}^{n} |F_i|) \neq |S|$ **then**
  produce warning
end

Algorithm 1 focuses on execution *flows* (i.e., groups of instructions executing on target resources). *Flows* require resources to progress. However, all resources may not be accessible (depending on the parameters set by the user and the global system environment). For this purpose, we define R_F as the entire set of hardware resources that the *flow* F can access during execution. Taking as input these flows and the corresponding available resources, the algorithm iterates on each *flow* and creates a set containing all *resources* accessible by all *flows*. Then, if the cardinality of this resulting set S (i.e., number of available resources for the application) is different from the sum of all *flows* cardinality, the resource reservation may be suboptimal.

For example, we may execute a 2-process code with four cores per process on an eight-core node. If both processes spawn more than four threads, resources will be overloaded. In this case Algorithm 1 would produce a warning informing the user that processes created too many threads according to available resources. Similarly, if processes created less than four threads, Algorithm 1 would produce a warning about idle resources.

The previous algorithm alone is not enough to detect all situations where different resource usage may appear. Thus, Algorithm 2 focuses on resources. It traverses through each resource R and checks, for each flow F, if R is included in

Algorithm 2: Resource-Centric Algorithm

```
Data: Flow = F_0, ..., F_n, Resource = R_0, ..., R_n
foreach R ∈ Resource do
    S ← ∅
    foreach F ∈ Flow do
        if R ∈ R_F then
            S ← S ∪ F
        end
    end
    if |S| ≠ 1 then
        produce warning
    end
end
```

the set of resources R_F that F can access. If more than one *flow* can access one specific *resource*, or if a *resource* can't be accessed, then the main repartition may be suboptimal.

For example if we launch a 2-process code with four cores per process on an 8-core node and disabled process binding, the job manager may allocate the same cores for both processes. It would lead to an execution scheduling eight threads on four cores, keeping four cores idle. In this situation, Algorithm 2 detects that four cores are used by two different processes and produces 2 warnings: one about overloaded resources and another one regarding idle cores.

We might note that depending on what *flows* are given when running algorithm 2, results may vary. Let us look at a simple examples to understand the subtle but important difference. The example is a multi-threaded application using only one process. Depending on the runtimes arguments, all the threads created could access all the resources available to the process. In this case, it would be the scheduler's duty to manage threads and resources efficiently. Now, when inputting threads as *flows*, algorithm 2 would detect that each resource is accessible by all the threads and would generate a warning. On the other hand if the process is inputted as a *flow* the algorithm would look at the number of threads created and the number of cores available. If the number of thread is equal to the number of cores, the algorithm would not detect a misuse and thus not generate any warning. In practice, this algorithm can thus be used on different granularities, when applicable The results from these granularities can also be combined. It could be of importance to know that the right number of threads has been created in the multi-threaded process, but that threads could be migrating and causing performance issues. Moreover, if information is

not available at one level or another, only applying the algorithm once will still provide valuable information. On the previous example, with only the process level granularity we know if the number of thread created was adapted to the available resources. If on the other hand we receive a warning telling that all the threads can access all the resources, we might investigate further. In this case we could, for example, determine that we want to pin each thread to a core.

For example by launching a multi-threaded process, each thread created might be able to access all resources allocated to the process. Thus, by inputting threads as *flows*, algorithm 2 would detect that each resource is accessible by multiple threads and would generate a warning. On the other hand if the multi-threaded process is inputted as a *flow* the algorithm would look at the number of threads created and if it is equal to the number of cores available, there are no warning to be printed. Thus, in practice this algorithm might be used on different granularities when applicable and results combined. Moreover, if information is not available at thread level, only applying the algorithm to the whole process will still provide valuable information.

These algorithms used together detect misuses coming both from resource allocation and flow repartition giving an accurate identification of the source of potential resource misuse.

According to configurations introduced in chapter taxonomy, Algorithms 1 and 2 mainly focus on spatial stacking. Indeed, when receiving a warning the user will have information on both the resource usage (idle or overloaded resources) and the spatial configuration (concurrent accesses). Even if such warnings are very valuable, the user will not be able to know when and where (inside the source code) the problem occurred. Adding some temporal analysis will produce far better reports, giving an idea where the application is competing for resources while removing false-positive outputs that a spatial-only analysis would produce. To apply temporal analysis, no new algorithm is required. It is actually done by decomposing the execution of an application in chunks where runtime stacking is consistent. We define a *consistent stacking timeslice* to be a runtime execution chunk where cpusets of all Flows remain unchanged. Each chunk or *timeslice* is then analysed by the `spatial algorithms`.

Algorithm 3 presents the full analysis process of our algorithms. We first determine all timeslices T_{app} of an execution, having a timeslice ends and a new one begins at each change in cpuset of any Flow. Then for each slice T_n we apply algorithms 1 and 2. We obtain a spatial analysis of each timeslice separately and thus a temporal and spatial analysis of our application.

Algorithm 3: Full analysis process

Data: $T_{app} = T_0, ..., T_n$, $Flow = F_0, ..., F_n$, $Resource = R_0, ..., R_n$

foreach $T \in T_{app}$ **do**
    Algorithm1 (Flow, Resource)
    Algorithm2 (Flow, Resource)
end

5.3 Tool producing logs

The algorithms presented in Section 5.2 take subsets of resource allocation as input. However, we needed to get the data from applications' executions to apply the algorithms. To produce and process these input data, we designed a dynamic tool which oversees the execution of an application and produces logs containing information about threads and runtime libraries. We called this tool the *Overseer*.

The goal of the Overseer is to gather relevant input data that will be used by our algorithms. It collects information on threads/processes from their creation to their destruction. We implemented it as a library which is preloaded at execution time through the `LD_PRELOAD` mechanism. Each process created during the execution of the target application loads its own instance of the library, so that the tool gets access to information about all processes.

We want first to retrieve information about thread placement. To perform this, the library wraps the `pthread_create` function to track all created threads and their parameters (pid, tid, name of the module calling `pthread_create`, cpuset the processes threads are allowed to run onto...). Now that the Overseer has access to information from all processes and threads, it can create log files. We create a file per process, each of them beginning with all information gathered on the process at startup. We chose this solution to limit contention on files and limit the overhead of the Overseer. This technique might not scale with a large number of processes but solution as aggregators like PADAWAN [47]. Then each `pthread_create` function call adds a line in the file with information on the new thread. In order to keep track of the cores a process has access to, the Overseer also wraps the `set_affinity` function calls adding a line in the log file. In addition, each line in the output trace file is written with a time stamp to provide a temporal view of the execution.

To reduce the overhead to a bare minimum we can focus on the information we want to collect and especially when we want to collect them. As the only information needed to create our execution chunks are beginnings and ends of parallel section of codes it is possible to wrap all functions, pragmas, and so on

creating parallel section (*#pragma parallel*, *#pragma parallel for* for OpenMP, *MPI_Init(...)*, *MPI_Finalize(...)* for MPI, ...). Although this solution is not intrusive for the user, it is difficult to implement as it needs to be exhaustive. This solution requires a lot of work which was not necessary for this thesis. It is however an interesting future work in the context of a future industrialization. An easier solution is to make specific calls to functions in the user code and wrap these calls with the Overseer. We can imagine that the user can perform the calls himself. To be less intrusive, these calls could be added by a source-to-source analysis of the code, or at compile time by the compiler, or even by binary instrumentation. In a desire for simplicity, we used the first solution: adding calls directly in the source code. This is the less elegant solution, but also the easiest to implement to perform proof of concept and tests. To apply our algorithms on *consistent stacking timeslices*, each time we encountered the end of a parallel region, we arbitrarily consider that the cpusets of this parallel region thread are emptied. This way, even if those threads are just asleep and not destroyed, we still consider that the runtime attached to this parallel region is not active, and should not issue a warning if its resources are used by other threads.

Note that it is possible to give the user a great deal of information on his application and runtime stacking with this information only. We can determine which runtimes are alive at each instant, determine which interactions are actually possible between runtime and discard those which are not, determine which runtimes are problematic and where these parallel sections are situated in the code. With all this information, debugging and optimising runtime stacking becomes possible, when it was previously almost impossible.

Information provided by these logs is not exhaustive, however it allows the analysis of interactions of runtime libraries with our algorithms presented in Section 5. Moreover, the small amount of data collected leads to a lightweight tool that generates small traces and therefore a small overhead.

5.3.1 Other trace possibilities

Even though we judged the information provided by the Overseer sufficient for further analysis, it would also be possible to get perfect placement and scheduling information. Indeed, it is theoretically possible to monitor every scheduling event during an application's execution. This information is found at kernel level, in the scheduler. Every time a thread is scheduled on a core, a function could be called, or a callback produced to launch our monitoring function. However, the number of scheduled events could be too big to be practical. By interfacing with the scheduler, we would intercept each and every thread movement. Every time a thread from the system is scheduled we would

produce logs. All these manipulations would interfere with the application. In the end the application behaviour could be changed and not represent a run of the application without the instrumentation. Last but not least, getting this information would mean modifying the kernel. The problem is that this action is not without consequences. Especially on a cluster used by multiple users. It is not possible for every user to modify the kernel as he wishes. Thus, we would have needed to launch applications in virtual machines, implementing our custom kernel. This would also change the behaviour of applications. And in the end, pushing or approach into a cluster would be impossible. No cluster administrator would consent (and would be rightly to oppose) to use a custom, non monitored and updated kernel modification.

An other strategy could be to use sampling techniques to monitor the threads. For example, by looking at */proc/[pid]/task/[tid]/status* file (on a Linux system) of each thread involved in the computation we can determine their states. We can find which ones are running, blocked, waiting of terminated, and also the number of context switches, the number of blocked signals and so on. By looking at these files at regular intervals, we can get an accurate overview of how threads are behaving. This solution can be implemented and set up without perturbing the development and compilation chains of the target application. However, this technique has two major problems. The first one is that it is intrusive. The more precise we want the data, the more often we need to look at the files. This would utilize compute resources and more importantly change the behaviour of the application. Indeed, by slowing the execution, or by interfering with the threads in any way, we may change the behaviour of the application. This would result in observations diverging from non sampled runs. The second problem is that sampling does not give perfect information. We could miss important information as we only get scattered data points and not a constant view of the thread's behaviour.

We tested this sampling technique in the beginning stages of the PhD. Unfortunately, we found out that the overhead induced was too high. Getting one sample point every second could add as much as a 30% overhead on small benchmarks. Note that getting this little number of data point would not be precise enough for our purpose. Moreover, the overhead is almost already to high for the technique to be by simulation codes or benchmark tests.

5.4 Putting it all together: analysing logs and producing warnings

Now that we have logs of runtimes lifespan, threads creations and resource usage in general we can apply the algorithms. We designed a second tool that

takes the logs produced by the Overseer, parse them and produces an analysis As noted in the previous section, the logs are not exhaustive. By looking at processes' allocated resources we can only speculate on the real resource usage. However, if a process created ten threads on one core, we know for sure that it was overloaded. An uncertain example would be two runtimes both using the same whole four-core node, and each creating two threads. They could use disjoint resources, or use the same. This is where the scheduler's job comes into play.

Moreover, some detected misuse could be a feature of the code. For example some simulation need a lot of memory per thread. More memory than available to one core. It is then possible to allocate more than one core per thread. The thread will 'waste' one core's compute resources but will have access to enough memory to perform its tasks. With this kind of configurations, our analysis would detect a lot of unused cores. But it is the way the simulation was meant to use the available resources. Thus, our analysis products can only be warnings and not hard errors. Note that the analysis still gives important insight for these configuration cases. Indeed, when using these allocation schemes, the algorithms will confirm or not, that the desired behaviour was achieved.

By parsing the output traces, the post-mortem tool retrieves all the relevant information about processes and threads. Algorithms 1 and 2 are then applied. *Resources* are assimilated to cores and *flows* to processes and their threads. As presented in Section 5, we run algorithm 2 twice, once at the process granularity and once at the thread granularity. With all information gathered from traces and algorithm result, an output is generated, giving a summary of process cpusets and thread placement (*spatial stacking configurations*) as well as information about resource usages and runtime libraries through warnings. With this information and hints, user can determine if a better resource usage is possible and how to achieve it. Listing 5.1, shows a simple output when using only one process on a four core node. In this case we get a warning telling that some resources are not used. The solution could be to use more processes on the node, or create threads in the process.
Listing 5.2 shows the results the first modification. This time four processes are creating, one on each core of the node. We don't get any waning. Listing 5.3 shows what happens with the second modification. We used created three more threads in the process. We receive a warning informing us that threads might be migrating from core to core. However, if we use launch the analysis at the thread granularity (and imagine that we pinned the threads) we get the results presented in Listing 5.4. This time we see that each thread can only use one core and which results in no warning from the algorithms.

Listing 5.1 – Algorithm output: 1 single-threaded Process on a 4-core node

```
+ Node 0: 4 cores available, 1 process created.
          Process 0 created 1 thread on cpu(s) [0]
  ## WARNING ## Cores [1 - 3] may stay idle during execution (underloaded
      resources)
Analysis conclusion: 1 warning(s)
```

Listing 5.2 – Algorithm output: 4 Processes on a 4-core node

```
+ Node 0: 4 cores available, 4 process created.
          Process 0 created 1 thread on cpu(s) [0]
          Process 1 created 1 thread on cpu(s) [1]
          Process 2 created 1 thread on cpu(s) [2]
          Process 3 created 1 thread on cpu(s) [3]
Analysis conclusion: 0 warning(s)
```

Listing 5.3 – Algorithm output (Process granularity): 1 Process with 4 threads on a 4-core node

```
+ Node 0: 4 cores available, 1 process created.
          Process 0 created 4 thread on cpu(s) [0-3]
  ## WARNING ## Cores [0 - 3] may be used multiple thread at the same time (
      overloaded resources)
Analysis conclusion: 1 warning(s)
```

Listing 5.4 – Algorithm output (Thread granularity): 1 Process with 4 threads on a 4-core node

```
+ Node 0: 4 cores available, 1 process created.
          Process 0 created 4 thread on cpu(s) [0-3]
                Thread 0 created on cpu(s) [0]
                Thread 1 created on cpu(s) [1]
                Thread 2 created on cpu(s) [2]
                Thread 3 created on cpu(s) [3]
Analysis conclusion: 0 warning(s)
```

5.5 Experimental Results

5.5.1 Test bed description

The test bed used during this Thesis is composed of a cluster of Intel Sandy Bridge, Intel Haswell and Intel KNL nodes. Sandy Bridge are 2-socketed nodes, with 8 non-hyperthreaded cores per socket for a total of 16 cores. Haswell nodes are also 2-socketed but possess twice the number of cores, that is 16 cores per socket and a total of 32. These cores are also using the hyperthread technology. Each core is composed of two hyperthreads. Last but not least, the KNL are composed of 68 cores. Among those cores, 4 are reserved for the OS, leaving 64 compute physical for the applications. Each of these core is hyperthreaded with 4 hardware threads for a total of 256 hyperthreads for the application.

		2 Haswell nodes 64 MPI tasks 2 OpenMP threads per task	32 Haswell nodes 64 MPI tasks 32 OpenMP threads per task	1 KNL nodes 64 MPI tasks 4 OpenMP threads per task	8 KNL nodes 64 MPI tasks 32 OpenMP threads per task
Lulesh	with Overseer	46.945	2672.100	398.580	213.760
	without Overseer	46.790	2668.745	398.135	213.695
	overhead (%)	0.33	0.125	0.11	0.03
miniPE	with Overseer	18.805	303.733	68.350	102.331
	without Overseer	17.789	303.325	66.728	102.272
	overhead (%)	5.71	0.13	2.43	0.06
AMG	with Overseer	34.532		65.553	
	without Overseer	33.476		64.939	
	overhead (%)	3.15		0.95	
Nekbone	with Overseer	29.219	13.217	98.629	41.174
	without Overseer	28.255	12.623	98.159	40.920
	overhead (%)	3.05	4.5	0.48	0.62

Table 5.5.1 – Evaluation of the Overseer overhead on various architectures

5.5.2 Tool overhead

Table 5.5.1 presents the overhead of the Overseer when running along with CORAL benchmarks on different hardware configurations. The first two columns show the results on Haswell nodes, and the following two present results using Intel KNL nodes. The first column presents the results on two Haswell nodes, using sixty-four MPI tasks (thirty-two per node), and two OpenMP threads per task (two per hyperthreaded core). The second presents the results of runs using thirty-two Haswell nodes, using sixty-four MPI tasks (two per node, one per socket), and thirty-two OpenMP threads per task (using all hyperthreads of a socket per task). The third column shows results using one KNL node, sixty-four MPI tasks and four OpenMP threads per task. The last column presents results using eight KNL nodes, sixty-four MPI tasks and thirty-two OpenMP threads per task.

This table shows a low overhead when using our Overseer. In fact, the tool only instruments a couple of functions. These functions are related to thread creations, destructions, and cpuset modifications. As most of the time in HPC applications, cpusets of processes are set once and for all at their creation and threads are used by multiple libraries without them being destroyed and re-created, the overhead of the Overseer stays low. Thus, the interference of our tool can almost only be seen at startup. We can observe this in the table: the longer the execution the smaller the overhead percentage is. For a 20 seconds execution, the 6% overhead seems high, but for a more representative time (several minutes), the overhead drops and becomes negligible.

5.5.3 Benchmark Evaluation

In order to show how resource allocation and resource placement have an impact on the performances of applications, we ran four CORAL benchmarks

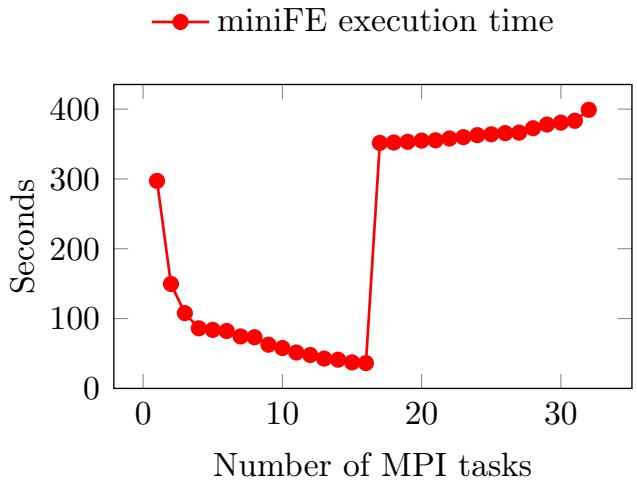


Figure 5.5.1 – Execution time of miniFE with different number of MPI tasks

with different allocation and placement methods on our test bed. Figure 5.5.1 presents the total execution time of the miniFE application on a Sandybridge node (16 cores without hyperthreading) when the number of MPI tasks grows and the problem size remains the same. We can see that from 1 to 15 cores while the node is underloaded, the execution time of the simulation decreases. It reaches its best time when the node is full, at 16 MPI tasks, and then execution time starts to increase when we overload the node with more MPI processes than available core (from 17 to 32 ranks). We can conclude that better performances are obtained when using the right number of cores which is obvious. However, finding the best configuration is not always easy, and it is often hard to identify the source of a lack of performances when the error comes from allocation and placement of tasks and threads. Indeed, neither error nor warning are produced by the compiler or at execution.

This experiment shows that the resource allocation has an impact on the performances of a simulation. Our algorithms presented in Section 5.2 can help find a resource allocation configuration using the maximum of resources. For example, using our tools with each configuration from Figure 5.5.1 produces warnings except when using 16 tasks, which is the configuration using the architecture at its fullest without overloading the node. With an underloaded node, e.g., with only one MPI task, the analysis produces the following warnings:

```
+ Node 0: 16 cores available, 1 process created.
          Process 0 created 1 thread on cpu(s) [0]
  ## WARNING ## Cores [1 - 15] may stay idle during execution (underloaded
      resources)
Analysis conclusion: 1 warning(s)
```

In the same manner, an execution overloading nodes with 32 MPI tasks outputs the following warnings:

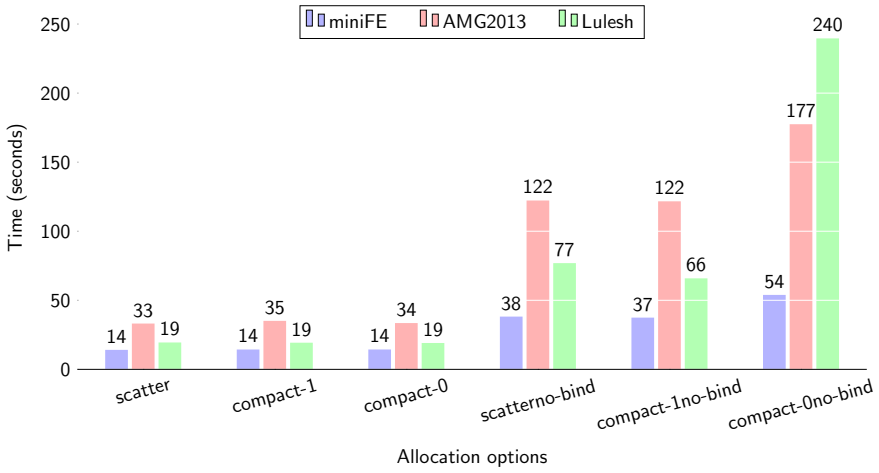


Figure 5.5.2 – Execution time of CORAL Benchmarks on Haswell nodes

```
+ Node 0: 16 cores available, 32 process created.
          Process 0 created 1 thread on cpu(s) [0]
          Process 1 created 1 thread on cpu(s) [1]
          [...]
          Process 16 created 1 thread on cpu(s) [0]
          Process 17 created 1 thread on cpu(s) [1]
          [...]
  ## WARNING ## Cores [0 - 15] may be used by more than one process/thread (
      overloaded resources) (spatial-concurrent configuration)
Analysis conclusion: 1 warning(s)
```

Figures 5.5.2, 5.5.3 and 5.5.4 present the impact of tasks and threads placement on application performances. For these experiments we used three CORAL benchmarks: Lulesh, minFE and AMG2013, and three target architectures: Haswell nodes with hyperthreaded cores, Sandybridge nodes without hyperthreading, and one KNL node with four hyperthreads per core. For each application, we varied the runtimes options. The 'scatter', 'compact,0' and 'compact,1' correspond to Intel OpenMP 'KMP_AFFINITY' options. The 'scatter' option means that all OpenMP threads are spaced as much as possible on cores (to maximize cache size and memory bandwidth), the 'compact,0' places one thread per logical core (including hyperthreads), and the 'compact,1' option fills physical core first. The 'no-bind' option indicates that process binding was disabled in the job manager. In our case, it means that all processes are given the same cpuset, resulting in overloaded resources and idle ones. Figure 5.5.2 shows the results of each benchmark using two Haswell nodes, eight MPI processes (two per node) and sixteen OpenMP threads per process effectively using all cores and all hyperthreads of both nodes. Figure 5.5.3 exposes the results of the same benchmarks on two Sandybridge nodes, eight MPI processes (two per node) and eight OpenMP threads per process effectively using all cores of both nodes as hyperthreads are not activated. Finally,

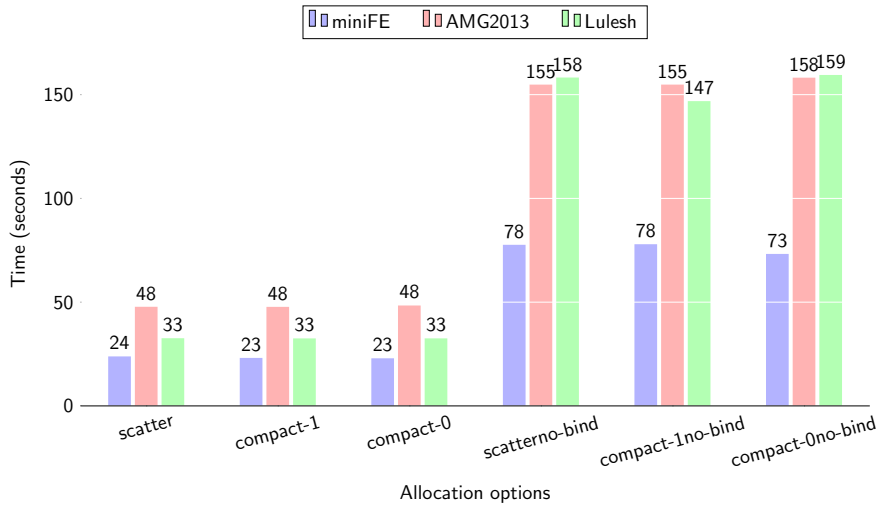


Figure 5.5.3 – Execution time of CORAL Benchmarks on Sandybridge nodes

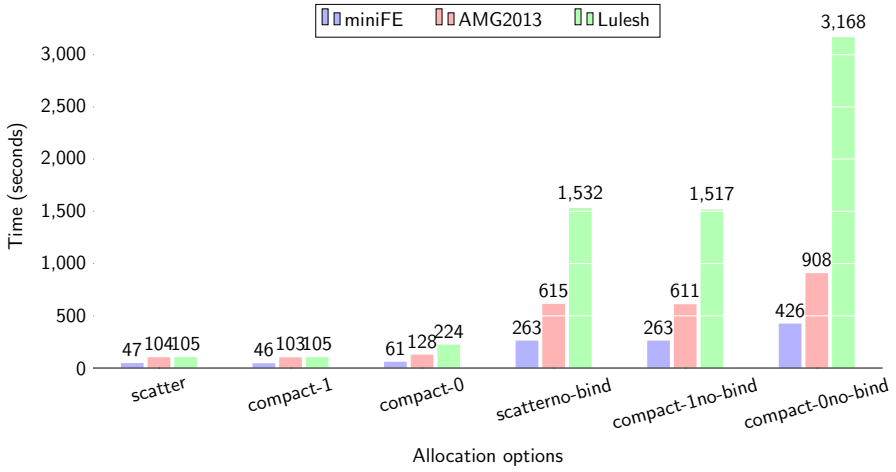


Figure 5.5.4 – Execution time of CORAL Benchmarks on a KNL node

Figure 5.5.4 presents the results of the benchmarks on one Intel KNL node with eight MPI processes and eight OpenMP threads per task using all cores but half of the hyperthreads of the node.

These graphs exhibit two behaviours. First when the process binding is disabled, execution time increases on all architectures. This was expected as we use fewer cores for the same computations and the same number of processes/threads. By disabling the binding the number of threads per core highly increases seriously impeding performances. This is especially strong on KNL nodes which use a lot of hyperthreads. Note that in these cases, our analysis produces the warnings to the user about a clumsy use of resources:

```
+ Node 0: 32 cores available, 4 process created.
          Process 1 created 8 thread on cpu(s) [0-8]
          Process 2 created 8 thread on cpu(s) [0-8]
          Process 3 created 8 thread on cpu(s) [0-8]
          Process 4 created 8 thread on cpu(s) [0-8]
  ## WARNING ## Cores [0 - 8] may be used by more than one process/thread (
      overloaded resources) (spatial-concurrent configuration)
  ## WARNING ## Cores [9 - 31] may stay idle during execution (underloaded
      resources)
Analysis conclusion: 2 warning(s)
```

Furthermore, we can see that the OpenMP environment may have an impact on performances. For example, on Haswell nodes (figure 5.5.2), using the 'compact,1' option with the AMG2013 benchmarks adds a 6% overhead to the execution time compared to the same execution with the 'compact,0' option. On the other hand, running the Lulesh benchmark using the 'compact,0' option on KNL nodes (figure 5.5.4) more than doubles the execution time compared to the same execution with the 'scatter' option. These results are explained by the fact that these options change the placement and binding of threads. As all cores and hyperthreads are used with the AMG2013 benchmark on Haswell nodes, the overhead observed is probably a consequence of a different data placement intensifying the NUMA effects when using the 'compact,1' option. Note that in this case, our analysis do not produce warnings as resources are busy with one execution flow. Moreover, this placement will not always influence performances in the same manner on all architectures. For example the same benchmark on Sandybridge nodes (figure 5.5.3) exposes the best performances with the 'compact,1' option. By looking at the analysis output, users could see that the cpusets of processes are spread on multiple processors but the conclusion regarding performances would come from knowledge of both architecture and application's data usage. However, when using these OpenMP options with a partially full node, the analysis detects what may be wrong use of resources and produces warnings. For example using the 'compact,0' option on a KNL node more than doubles the execution time of the Lulesh benchmark (figure 5.5.4). In this case the tool would detect that only half the cores of the node are used and inform user that half the resources may stay

idle during execution. Note that with the 'scatter' or 'compact,1' options the analysis would also produce warnings this time informing that all cores are used but with only half the hyperthreads.

These experimental results show that allocation and placement of tasks and threads have an impact on application performances. This impact may vary depending on architectures and applications. Our Overseer and the analysis it enables can give user a view of resource usage and thread placement, as well as it produces warning when a potential misuse of resources is detected. This kind of analysis is important as allocation and placement are not checked by application or system, even when obvious misuses occur, which may lead to bad performances.

5.5.4 Improving analysis with temporal information

The analysis previously presented only takes spatial configuration into account. That is to say, each time a thread is created in the application, it is considered alive for the whole remaining execution. If this kind of analysis can already greatly help users to optimise their code, it can also create inaccurate or plain wrong reports. Adding a temporal view of the application may help users better understand codes and runtime stacking as well as eliminate analysis errors. Indeed, in some cases, a spatial-only analysis may create false-positive warnings that a temporal analysis could avoid. To take temporality into account, we split executions into timeslices, each timeslice defined by a portion of code where cpusets stay unchanged.

```
int main(int argc, char ** argv) {
    MPI_Init(&argc, &argv);

    initialize_variables();

    omp_solver();
    tbb_solver();

    MPI_Finalize();
    return 0;
}
```

Listing 5.5 – Hybrid MPI-OpenMP-TBB code

Let's consider the code in Listing 5.5. To get a simplified view, this code only makes one call to an OpenMP optimised function and then one call to a TBB optimised one. On an 8-core node, the spatial-only analysis with our tools would produce the following report:

```
+ Node 0: 8 cores available, 1 process created.
          Process 0 created 16 threads on cpu(s) [0-7]
  ## WARNING ## Cores [0 - 7] may be used by more than one process/thread
                (overloaded resources)
                (spatial-concurrent configuration)
Analysis conclusion: 1 warning(s)
```

We get a warning because the number of threads created is twice the number of available cores. However, by dividing our execution into timeslices where runtime stacking is consistent we would get the following report:

```
+ Node 0: 8 cores available, 1 process created.
     Process 0 created 16 thread on cpu(s) [0-7]
       - Chunk 1: 8 threads created on cpu(s) [0-7]
       - Chunk 2: 8 threads created on cpu(s) [0-7]
Analysis conclusion: 0 warning(s)
```

This report informs that indeed there were more threads created than available cores, yet the resources never got overloaded as runtimes were not running concurrently. To highlight the significance of a temporal analysis for our tools, we implemented micro-benchmarks. These codes extract the minimal configuration needed to investigate temporal interactions using our analysis algorithms.

Thus, the temporal analysis here allows determining more precisely what happens at execution time while also removing false-positive warnings. Likewise, on Listing 5.5 code, exploiting only half the available cores per runtime would lead to the following reports:

```
Report without temporal analysis:
+ Node 0: 8 cores available, 1 process created.
          Process 0 created 8 thread on cpu(s) [0-7]
  [...]
Analysis conclusion: 0 warning(s)
```

```
Report with temporal analysis:
+ Node 0: 8 cores available, 1 process created.
          Process 0 created 8 thread on cpu(s) [0-8]
   - Chunk 1: 1 threads created on cpu(s) [0-7] (mpi)
       ## WARNING ## Cores [0 - 7] may stay idle
                     (underloaded resources)
   - Chunk 2: 4 threads created on cpu(s) [0-3] (OpenMP)
       ## WARNING ## Cores [4 - 7] may stay idle
                     (underloaded resources)
   - Chunk 3: 4 threads created on cpu(s) [4-7] (TBB)
       ## WARNING ## Cores [0 - 3] may stay idle
                     (underloaded resources)
Analysis conclusion: 3 warning(s)
```

Without using temporal analysis, the tools would not always detect misuse of resources. Indeed, it would detect that the number of threads created is the same as the number of cores available which is seemingly a good behaviour. Depending on the cpusets used by the OpenMP and TBB runtimes, the analysis could detect a spatial concurrency. On the other hand, by adding the temporal analysis we see that at most only half the cores are used during the execution regardless of thread placement. Temporal analysis can thus also detect resource

misuse that could not be seen previously. In our example, it is clear that resources are not used at their maximum efficiency. A user could with a glance at the analysis' report determine if resources are used as he intended to, and then determine if he wants to add threads in parts of the code. We can note that we also get a warning when the application is setting up and only the MPI runtime is alive. As the creation of TBB threads is not done instantly after the end of the OpenMP section, we detect that MPI is alone here as well and the analysis should produce a warning. We still need to determine if we want to overload the logs with all these warnings. On the one hand this creates longer analysis files and hinders comprehension. On the other hand this information is useful in the case of threads regularly getting created and destroyed which could be optimised by recycling threads.

```
int main(int argc, char ** argv){
    MPI_Init(&argc, &argv);
    initialize_variables();

    while(variable){
      omp_solver();
      tbb_solver();
    }

    MPI_Finalize();
    return 0;
}
```

Listing 5.6 – Hybrid MPI-OpenMP-TBB code

The next example, presented in the Listing 5.6, shows what happens when more parallel sections are launched by the code. Usually the OpenMP and TBB runtimes recycle their threads, i.e., threads of the OpenMP and TBB runtimes will be spawned only once and not at each loop iteration. When temporal analysis is turned off, we get the exact same report as with the Listing 5.5 code, as we only see one spawn phase for OpenMP and one for TBB. By turning the temporal analysis on, we see the alternating runtimes, and we do not detect concurrency between runtimes.

To sum up, temporal analysis helps determine if runtimes are actually concurrent. This improvement detects more resource misuses, and discards some false positives warnings. Indeed, two spatially concurrent runtimes can be temporally independent which will not cause resource misuses but would generate a false positive warning with the basic analysis. Spatially and temporally independent runtimes may not generate warning with the basic analysis but leave idle resources.

5.6 Conclusion and Discussion

This chapter presented algorithms that analyse resource usage of an application. We developed a post-mortem tool that implements these algorithms and provides the analysis as well as warnings to the user. In the meantime, we had to develop another tool, the Overseer, used in parallel to application's execution. It collects all the information needed for the analysis. As we explained, our choices led to a lightweight tool, that can be used anytime without changing the development process of an application (source modification, or compilation chain additions). Moreover, the hints and warnings produced give information about potential misuse of resources. This kind of analysis was not possible before.

Our analysis could still be greatly improved on. We have some future work planned. First, we only looked at basic configurations involving threads and cores. A huge improvement would be to include more resources into the analysis. Indeed, the algorithms can be applied to any kind of resources like memory, I/O buses, network cards and so on. With current architecture, applying the algorithms to memory and data seems to be the most beneficial upgrade. Looking at data placement would decrease NUMA effects present on NUMA nodes used on most current supercomputers. By all means placing data near the threads using them is bound to reduce data transfer time and latency. Looking at thread placement in regard to I/O buses and network cards should become more and more important. Indeed, as explain in Chapter **chap:context** future architectures will certainly be so heterogeneous that some cores could be closer to these resources than others. Having a specific thread making communication with other nodes, and placing it near the network card would reduce communication time.

Second, we plan to improve our analysis tool to provide optimal resource reservation arguments depending on runtime stacking categories and desired configurations. User usually know what they want to do: it could be using all the cores with one thread per core, use one process per socket, use all memory from a node with only one thread and so on. Now the difficulty is in conveying this information to the machine, using allocation options and runtimes arguments. We plan to design an abstraction for the user to describe the configuration he wants his application to run. Then our tool would determine the command line arguments to reserve resources, as well as the arguments and options to give to each runtime.

Lastly, we believe that this analysis could be coupled to already existing profiling tools. We already discussed in Subsection 5.5.3 that sometimes using all the resources without overloading them is not enough. Sometimes, a subop-

timal will amplify NUMA effects for example. A profiling tool like PAPI [48] that looks at the performance hardware counters would detect cache misses and other detrimental behaviour. Coupled with our analysis, it would determine which configuration to apply to remove these behaviours or limit their effects. Performance analysis tools such as Scalasca [41] or TAU [42] for example, could provide data or information to our analysis. These tools are designed to pinpoint the scalability bottlenecks like communications and synchronizations. Knowing which thread created a bottleneck could help improve thread placement. This would request more research and analysis of the causes and consequences of these negative effects regarding the runtimes and resource usage.

Using the results from this tool, the next chapter presents a dynamic `hypervisor` that we developed to manage threads and resources at execution time. This program uses the information dynamically gathered from the Overseer, and applies the algorithms to determine and manage thread placement.

Chapter 6

Hypervising Resource Usage

The previous chapter illustrated the impact of resource usage on application performance. Multiple factors are to take into account: resource allocation, and flow placement. Allocating too little resources will lead to over subscription and resource sharing which will, in turn, create cache thrashing and other hardware issues. Allocating too many resources in the other hand will 'waste' them, as more flows may have been used on these resources, or as another application could have used them. Flow placement, the second factor, is also important. Even if the allocation was just perfect, flow placement can ruin performances. Once again it is possible to oversubscribe or undersubscribe resources which would lead to decreased performances. An other factor to take into account is communications between flows. Placing flows using shared memory on the same NUMA node will generate less latency for example. Usually threads from the same runtime will most likely share information. Threads created from the same OpenMP call for example are very likely to use the same data, or very close data objects.

It is then necessary to consider all the components involved in such resource usage: the job/resource manager, the target compute-node configuration and the runtime implementations of parallel programming models. Thus, capturing a global view of resource usage is required to adapt the process/thread placement. That is why our approach called *Overmind* aims at providing algorithms overseeing resource requests to automatically take the best decision and enhance the use of each compute node inside a supercomputer. It uses the algorithms as well as implementation made in the Overseer presented in Chapter 5.

6.1 Bibliography

With the increasing complexity of HPC clusters, managing resources and threads is an important challenge [49]. Hybrid programming [50] as well as topology awareness [51] can have a significant impact on performances. Different

approaches exist to manage the execution flows and optimize their placement.

First of all, some programming languages and specifications have been created to exploit the benefits of both distributed and shared memory. For example, the Partitioned Global Address Space (PGAS) approach (UPC/UPC++ [52], OpenSHM [53], ...) enables a fine mixing between communication and concurrency. In such case, the underlying runtime has a global view and can manage process/thread placement as needed. Other initiatives like PaRSEC [54], StarSS [55], OMPSS [56] or StarPU [57] manage the scheduling of tasks over heterogeneous architectures. Once again, the global view of the tasks, workers and hardware topology helps improving process/thread placement and enhances resource usage. In such cases, our approach may not be helpful if the overall execution flows are taken into account. But if another model is added or if the runtime implementation relies on different independent libraries, our hypervisor may help optimizing the overall resource usage.

Other approaches deal with an integration of various programming models. For example, the Common Component Architecture [58] hides the software complexity with the help of components. However, if such components are relying on different parallel programming models and external runtime systems, management of execution flows can be complex. In such cases our approach can help optimizing the resource usage across components. Going further into model integration with regular standards is the unification of models. Some work propose ideas to improve the deployment of application layouts on compute nodes [59]. On the other hand, frameworks like MPC [20] provide a unified MPI/OpenMP layer designed to improve the scalability of HPC applications. MPC internally deals with the thread/process placement based on its global view. But this is limited to the MPI and OpenMP models. Adding another language to the parallel application will not benefit from this placement optimization while our hypervisor will take this new model into account and include the additional execution flows for resource-usage enhancement.

Finally, research is conducted to optimise thread placement and resource usage as a whole [60]. An interesting research axis is the study of memory locality [61]. Mapping algorithms like mpibind [62] are even designed to map applications based on the memory components and not the processors and cores like usual approaches. Following the memory architecture improves overall performances by providing as much cache and memory as possible to each application worker while also taking thread locality into account. The problematic of thread placement is well studied [63][64][65]. Placement algorithms and politics are numerous and vary depending on the goal to achieve. Some try to optimise application performances while other may want to lower energy consumption or limit overhead brought by communications and

synchronisations.

6.2 Overmind's Design

Our approach consists on a tool hypervising resource usage. It constrains the worker scheduling on a subset of available resources by capturing each active thread/process during the application execution. Thus, it is mandatory to have a perfect knowledge of all resources and their topology. Furthermore, since the number of workers can vary during the execution, the optimal layout may change for different phases. Therefore, our approach considers both the spatial (set of cores and their topology) and the temporal (application parallelism over time) aspects of resource usage.

To detail the need of such spatial and temporal approach, let us consider a target architecture containing N physical cores. If an application exploits N/2 threads during 90% of the execution time and N/2 additional threads for the remaining 10%, counting only the total number of execution flows is not enough. Indeed, there were a total of N threads for N available cores, but during the major part of the application execution, half of the resources were idle. Our approach would spread the N/2 threads on the available cores and eventually pack them together when the remaining N/2 additional threads are created. This method is even more useful when dealing with target architectures including SMT (i.e., logical cores). Indeed, exploiting physical cores might be the priority to avoid scheduling threads on logical cores. Hence, we must consider *temporal blocks*, like in Chapter 5 to better observe resource usage and to provide finer grain reorganization of workers. The worker behaviour shapes those blocks: when the number of active workers changes (e.g., workers being created, awaken, destroyed or going to sleep), a new block is created leading to a new placement.

Listing 6.1 – MPI/OpenMP Example Illustrating Temporal Blocks

```
1  int main()
2  {
3    MPI_Init();
4    ...
5    #pragma omp parallel for
6    for()
7    {
8      ...
9    }
10   ...
11   #pragma omp parallel num_threads(x)
12   {
13     ...
14   }
15   ...
16   MPI_Finalize();
17   return 0;
18 }
```

Listing 6.1 depicts an MPI/OpenMP pseudo-code illustrating such temporal block sequence. When execution starts, multiple workers are created (i.e., MPI processes) with at least one per compute node. These workers start the first block (lines 1 to 4) leading to a placement decision. Then, an OpenMP parallel region is opened creating additional workers (i.e., OpenMP threads), which may modify the resource usage. Thus, a new block is created (lines 5 to 9), generating a new global placement. When this region is closed, its associated workers become inactive, which changes the resource usage again. Therefore, a new block is created with the same number of workers than in block 1 (line 10). However, since the active workers may have been redistributed on the compute units, the new layout may differ from the one at the beginning of the application. Then, a new OpenMP parallel region is created with a different number of threads (`num_threads` clause), leading to a new block (lines 11 to 14). At the end of this region, the associated threads become inactive. We return to the previous number of workers, creating a final block (lines 15 to 18).

Because the number of execution flows (or workers) may vary from one block to another, it might be necessary to update the resource usage and worker placement in each block. Considering again Listing 6.1, we can imagine that the first OpenMP parallel region spawns as many threads as the number of cores. In such case, the workers will be spread on the available physical cores. Thus, OpenMP rank 0 will be on core 0, while rank 1 will be on core 1, sharing cache levels. If the second OpenMP region ask for more threads, the optimal placement will start using hyperthreads: rank 0 will be on the first hyperthread of core 0, and rank 1 will be moved to be located on the second hyperthread of the same core. Moreover, if another thread-based model, like Intel Threading Building Blocks (TBB), is spawned during the lifetime of the first OpenMP parallel region, one can imagine that the OpenMP threads will first spread on all available cores, and then be redistributed in a compact way to gather on all hyperthreads of adjacent cores to leave some available resources for the newly-created TBB threads. This would create a new temporal block to enable this resource-usage update.

In the end, the more information that can be gathered, the better the decisions. This information can come from the runtimes, the users, the resource allocator, or even from previous runs. However, it is also of paramount importance to be able to take good decisions with no a priori knowledge to handle any scenario of resource misuse. In this light, the first algorithm we propose for the Overmind does not take any prior information, and define the new worker placement only thanks to data it has collected during the current run. Further algorithms could take more information into account to better user resources and manage runtimes.

6.3 Thread management Algorithm

According to the main design described in the previous section, our approach is driven by the notion of temporal blocks and the resource-usage modifications across blocks. Thus, the resulting algorithms are divided into two categories: (i) the block detection and (ii) the intra-block resource usage.

First, let us look at block detection. Before anything else, it is necessary to consider the behaviour of a parallel execution as a set of temporal blocks managing different execution flows and, therefore, requesting various resource usage. To determine those blocks inside an application, our approach catches various operations regarding parallel programming. We linked the block creations and destructions to runtime libraries' functions. For example, with the MPI model, catching the initialization (`MPI_Init` function) enables the creation of the first block. The finalize function (`MPI_Finalize`) tells us when to close the block. Other blocks can be deduced from the thread behaviour coming from shared-memory models. For the OpenMP runtime for example, we need to detect the use of preprocessor pragma directives. As discussed before, one solution is to look through the OpenMP implementation to find which functions are called at the creation and end of parallel sections. Another solution is to instrument the code at compile time. The easiest solution to implement is to write an API and call it every time a parallel section starts or ends.

After determining the different blocks, the second point to take into account is the intra-block resource usage. Indeed, each time a block start or end, it is necessary to re-evaluate the resource usage. To design this second algorithm, we first studied worker placement with multiple MPI and OpenMP configurations with the help of our tool [25], described in Chapter 5. This tool, the Overseer, is checking if resources were misused. We found out that, most of the times, the desired configuration is as follows: MPI processes scattered as much as possible, preferably one per NUMA node, to enable better locality and larger memory bandwidth; workers spawned from the same shared-memory runtime as close as possible, to benefit from shared caches and common memory channels.

Algorithm 4 details the steps to compute *cpusets* of each process according to various parameters: the total number of processes located on the same compute nodes and the underlying hardware topology.

With this algorithm we get the workers of the same MPI process as close together as possible to benefit from data locality since these workers will use the MPI process data. Also, in one MPI process, we wish the workers from the same runtime to be together. They will most likely use the same set of data, so once again we want to benefit from data locality. Furthermore, since they are

Algorithm 4: Splitting available cores across processes

```
Input: process_number (value)
Input: node_number (value)
Input: process_id (value)
Output: new_process_cpuset = (set of values)
core_per_node ← get_core_number_per_node()
pu_per_core ← get_pu_number_per_core()
process_per_node ← process_number / node_number
local_process_id ← process_id % node_number
core_per_process ← core_per_node / process_per_node
first_process_core ← core_per_process × local_process_id
for core ← first_process to (first_process + core_per_process-1) do
  | new_process_cpuset ← new_process_cpuset ∪ {core}
end
```

using the same runtime, they will have the same resource usage and waiting policies. They should not interfere with each other, which is more likely with workers from different runtimes. If two workers from different runtimes are located on the hyperthreads of the same core, the waiting policy of the first one may hinder the performance of the other one. Moreover, since it is easier for the application developer, the code design may use different data sets for each runtime. The data of each worker will reduce the caching possibility of the other one, and even the meta-data from the runtime may hinder even more the data locality.

In addition to having the workers from a runtime close to each other, one may want them to be in order (e.g., OpenMP ranks 0, 1, 2 and 3 on cores 0, 1, 2 and 3 if no hyperthreads is necessary). Here again, this placement is based on the data locality. One example is the regular OpenMP `parallel for` construct: the `for` loop is cut into chunks handled by the threads. Most scheduling variants allocate chunks to each thread in a round-robin fashion. Hence, the first chunk containing the first iterations will be consumed by OpenMP rank 0, while the second one will be consumed by OpenMP rank 1, and so on. If memory accesses are quite regular, the contiguous OpenMP ranks will access contiguous data. It is then better to have them close to each other in order to benefit from data locality. Thus, Algorithm 4 is enhanced with a specific shrunk *cpuset* for threads located in each process according to the mechanism described earlier.

Finally, to handle unbalanced applications, this algorithm is extended to look at the notion of *thread load* and figure out if reducing the *cpuset* at the profit of another one may be beneficial to keep a good resource usage. This can be done at some specific points of the program performing sort of collective

operations to synchronize processes located on the same node.

6.4 Overmind's Implementation

We implemented the previous algorithms into our Overmind that can be used by any parallel application without source-code recompilation. Thus, it is available through a dynamic library that can be preloaded with the code binary and launched through any job manager (e.g., SLURM). Our implementation is decomposed into two distinct phases: (i) information collection and (ii) worker placement.

6.4.1 Information Gathering

The first part of our tool is called the *collecting phase* and is based on the Hwloc library [66] to gather the target architecture topology. To capture each thread information, we implemented wrapper functions for thread creations and destruction. By positioning the library just after the thread creation, we can gather all the required information. First, the Overmind's wrappers gather the target architecture layout and the topology information through the Hwloc library. It also gathers workers information such as *cpusets*, *IDs* and so on.

Now we still miss some important information: which library created which thread and resource allocation information that the user could have given to the resource manager. First, the resource reservation information can be extracted from the command line arguments. This information is on the node launching the job, which means we need a way to communicate this information back to the Overmind's instance running on compute nodes. Multiple options are at our disposition. One would be to implement plugins into the job manager to send messages to our Overmind. This solution could be the best one as we could get much more than this feature from plugins. For example if another job is running on the node our application is using, we could implement a plugins sending messages to the Overmind when this job ends. Then the Overmind could decide to use this free resources for the application still running. Obviously this option is the most costly to implement. Other options include creating temporary files to keep the information needed. This file would be in shared space for the launching node and the compute nodes. Lastly, the simpler option is to put this information into the application.

We implemented an API to get access to this information easily. Using a function from this API in the application call will trigger the wrapper which will collect all the information for our Overmind. This solution is maybe the one that is the closest to a 'quick and dirty' implementation, but it has the

merit of getting the information easily without failure. We judged that this solution was the best fitted to produce a proof of concept. Future work include replacing the API with an automated method that does not require user input.

We also used API wrapping functions to capture which thread was created with what library. For example, by putting an API function at the beginning of an OpenMP section will make every thread involved in this section call the function. By wrapping it, we can mark all the OpenMP threads from this parallel section. An other option to get this information would be to look at the callstack of the thread, the address of the thread creation function in the binary file, and look what library was loaded around this address. This technique was implemented in a test tool in the first year of the PhD. However, as we already used an API wrapped to get other information we did not implement this technique in the Overseer yet. Moreover, wrapping a function at each beginning of parallel section gave us extra information: the beginning of *temporal blocks*. By adding a last API wrapped function at the end of parallel sections, we have all the information we need to construct temporal blocks, and to know when to call algorithms to analyse and potentially change the placement of runtime and threads.

6.4.2 Worker Placement.

With the information gathered, we can now look at resource usage and determine if misuse are happening or if improvement can be made. This phase is also decomposed in two sub-phases. The first sub-phase finds a better worker placement while the second handles the actual changes in the *cpusets*.

For the first phase, the Overmind looks at each beginning and end of temporal blocks. And each time, it re-evaluates resource usage. It uses the algorithms described in Section 5.2. If no misuse is found, then there is nothing to do until the end of the block, or the beginning of a new one. On the other hand, if the algorithms detect a misuse, we go into the second phase and use the Overmind's Algorithm 4 to reorganise the processes' cpusets. To realize these changes, the Overmind relies once again on Hwloc, since it provides functionalities to change *cpusets* from processes as well as threads.

6.5 Experimental Results

We evaluated our approach to determine its overhead and to check if the performance are close to the best configuration of the application execution.

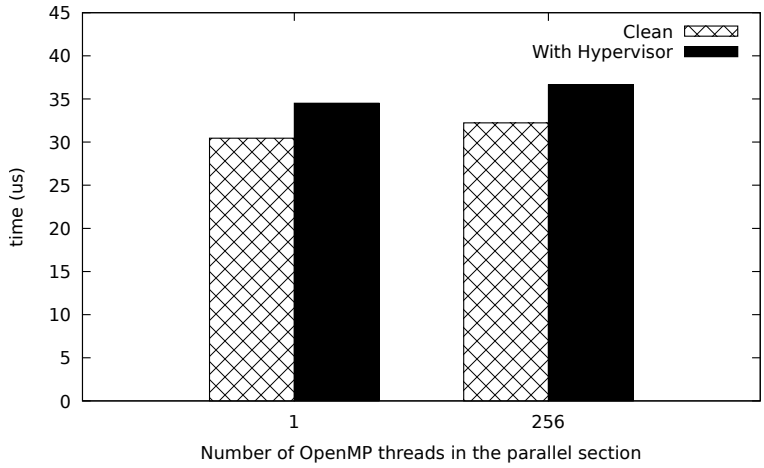


Figure 6.5.1 – Overmind overhead on one OpenMP region

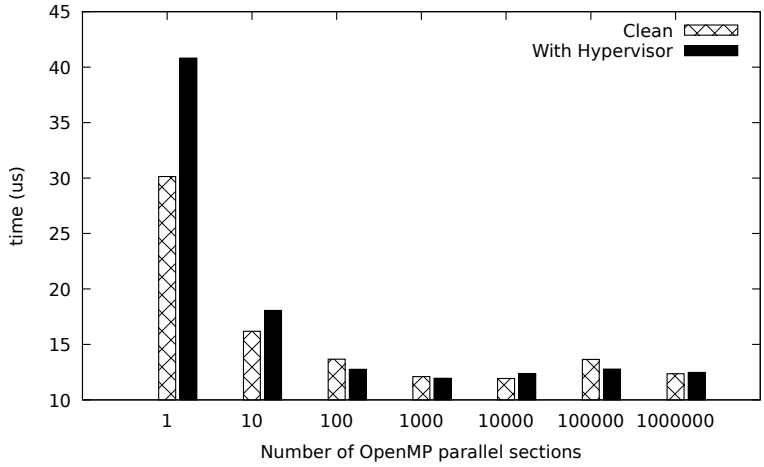


Figure 6.5.2 – Overmind overhead with increasing number of OpenMP regions

6.5.1 Overmind Overhead

To evaluate the overhead induced by the use of the Overmind, we performed two tests. Figure 6.5.1 illustrates the time of a simple OpenMP parallel region (1 and 256 threads) with and without the Overmind. The second test measures the mean time per parallel region when increasing the number of parallel constructs (Figure 6.5.2). For each parallel region, the Overmind applies the algorithm to find a good placement and redefine the cpusets. Since the number of threads remains the same, the threads are actually not moved. Hence, we measure the raw overhead of the Overmind. From these two experiments, we deduce that our Overmind has an overhead of 5 to 10us on the first OpenMP

parallel region (e.g., the first block to redistribute the threads), regardless of the number of threads. However, increasing the number of OpenMP regions does not linearly increase the overhead. In fact, the internal bookkeeping is done at the first block. Then, the only performed operations are computing the new placement and redefining the cpusets. With no thread movement, these operations are negligible, and the only visible overhead appears on the first block.

6.5.2 Benchmark Evaluation

We evaluated our approach with several CORAL Benchmarks (Lulesh, miniFE and AMG) on a small SLURM-based cluster with eight 68-core Intel KNLs. Among those 68 cores, 4 are reserved for the OS, leaving 64 compute physical cores for the application leading to 256 hyperthreads.

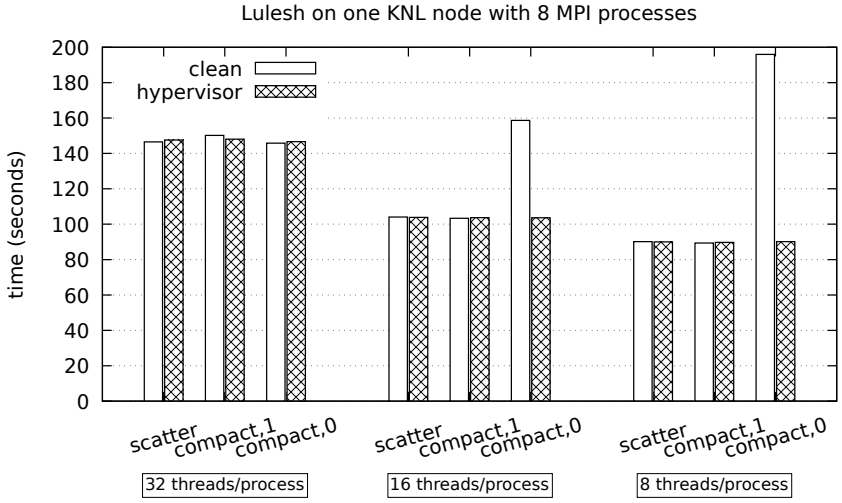


Figure 6.5.3 – Lulesh execution time with multiple configurations on 1 KNL node

The first experiment is on a single KNL node with 1, 2 or 4 threads per core. For each number of threads, we try three different placement policies: *scatter*, *compact,0* and *compact,1*. For each configuration (number of threads and placement policy), we evaluate the performances with and without our Overmind (Figure 6.5.3): white bars without the Overmind and plain bars with the Overmind. When there are enough threads to cover all available hyperthreads on the node (left cluster of bars in Figure 6.5.3), all configurations provide the same result. This is what we expected, since all compute units are used, without any oversubscribing, for the whole computation. However, when there are not enough threads to cover all available hyperthreads, the

placement policy *compact,0* is slower than the others. This one gathers the threads on hardware threads of the first cores of each MPI process, leaving half the cores unused, where the other policies place one thread per core. As explained in Chapter 2, hardware threads are not the same as plain cores. This policy makes use of less compute resources and may introduce cache thrashing issues. When enabling the Overmind on this placement policy, the misuse of resources is detected and the algorithm evenly distributes the OpenMP threads on all the cores. The measured performance are equivalent to the two other placements, removing the overhead due to the overloading of some cores.

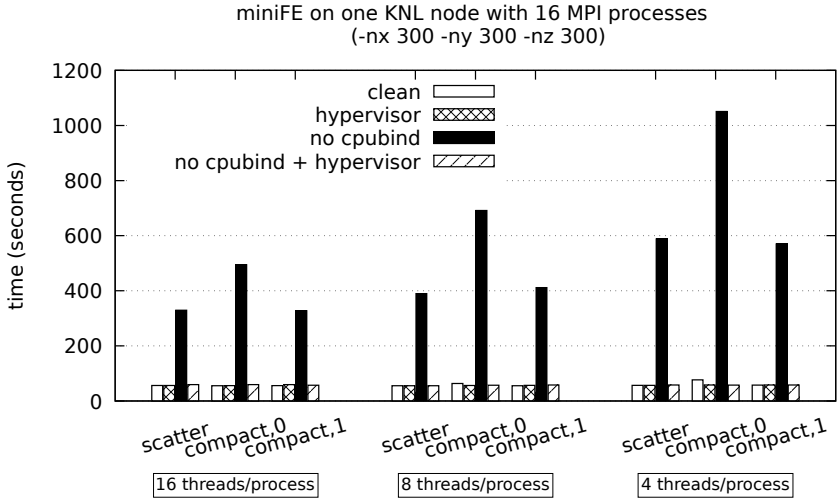


Figure 6.5.4 – Results of miniFE executions with multiple configurations on 1 KNL node

We realized the same kind of tests on miniFE with another configuration: 16 MPI processes, and either 4, 8 and 16 OpenMP threads per process. To check the behaviour of the Overmind when also MPI processes are not on a performing placement, we added the `-cpu-bind=none` parameter. This option removes the binding of processes by SLURM, and all the running processes are collapsed on the same core, with the same cpuset. In this specific case, MPI processes share the same computing resources and, since they will have the same behaviour for their OpenMP threads, these threads will also share the same computing resources. These results are displayed in Figure 6.5.4, with additional black bars for the configurations with `-cpu-bind=none` option, and the striped bars for the same configurations using the Overmind. When `-cpu-bind=none` is not used, we observe similar behaviour as Lulesh. When this option is set, every configuration behaves poorly, with the *compact,0* policy still being the worst. However, when our Overmind is enabled, all performances are improved. In each case, the Overmind evenly distributes the MPI processes

with a specific cpuset for each, then their OpenMP threads thanks to their own cpuset. The Overmind placement retrieves the best performance for each configuration.

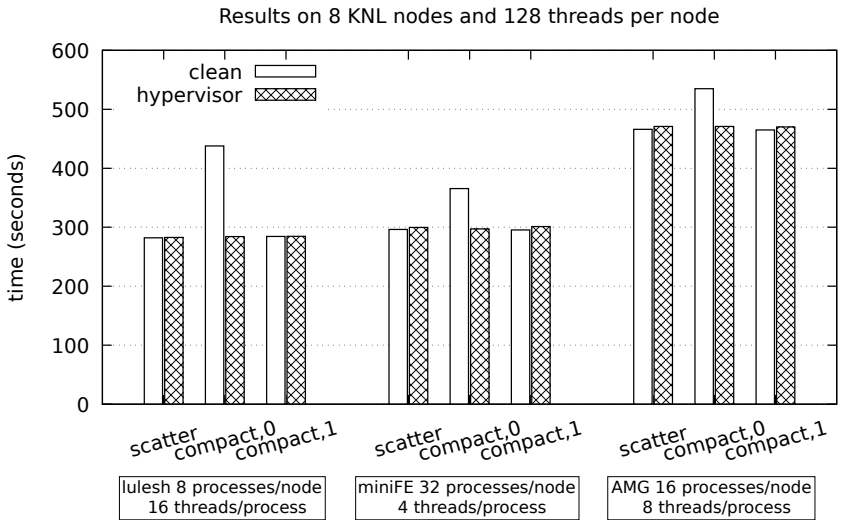


Figure 6.5.5 – Results of lulesh, miniFE and AMG executions with multiple configurations on 8 KNL nodes

Lastly, we executed the three CORAL benchmarks on 8 KNL nodes with multiple configurations. Except for one specific test on which the use of the Overmind slows down all configurations when a total of 256 threads per node are used and speeds up all configurations when a total of 64 threads per node are used, in most cases, the results are similar to the performances observed on 1 node. Some of these results are shown in Figure 6.5.5 for Lulesh, miniFE and AMG. We chose to display different number of MPI processes per tests to show that the performance of the Overmind does not depend on the number of processes and number of threads per processes. On all these configurations, we observe once again that the placement policy *compact,0* is the one providing the worst performance without the Overmind. Thus, the Overmind succeeds in correcting the problem and allows providing the same level of performance for each configuration.

6.6 Conclusion and Discussion

6.6.1 Conclusion

The approach presented showed that Runtime Stacking and resource usage can have a great impact on application's performances. As HPC architectures

and clusters evolve, getting more complex and more heterogeneous, we will see more and more runtimes alive at the same time and working with each other. Runtime interactions and Runtime Stacking issues in general need to be studied.

We re-used code, observations and experiences from the Overseer development to create the Overmind. This hypervising tool uses previous algorithms as well as new ones to manage flows and resources dynamically at execution time. It considers all threads/processes and map them efficiently onto the architecture topology. Available through a dynamic library, experiments show that the overhead of this method is quite low and it is able to dynamically adapt the execution-flow placement to reach better performance on different benchmarks (CORAL) and various architectures (from multicore to manycore).

The observation and hypervisor concept (Overseer and Overmind) presented can be the base for future research. The first part, the Overseer is the first step. Looking at what is happening is crucial to understand runtime interaction and get the opportunity to optimize applications. The second part, the Overmind is a proof of concept of what can be done. By using simple algorithms, it is possible to avoid catastrophic configurations where resources are badly misuse. However, this hypervisor can be greatly improved upon. New resource placement and reservation algorithms, new information gathered to get more insight on resource usage. Enhancements of temporal block detection to enable more flexibility when dealing with non-balanced workloads and more parallel programming models. Moreover, all the study we conducted were looking at specific resources: compute resources, core and hardware threads. It could be extended to other resources: GPU, Accelerators in general, but also I/O resources and so on. With architectures becoming more and more specific, we could have threads closer to the network card than other, thus placing threads communicating with other node near the cards would be a great choice.

6.6.2 Discussion

The approach presented works as 'stand alone' products. However, the more elements of the HPC chain are involved, the more precise and efficient the optimization. We already discussed this issue before when we could have modified the kernel to gather scheduling information. This information would be of great help to determine which runtimes are running at what time, when threads are in waiting state and so on. On the hypervising phase, we could imagine all the links in the chain of the resource reservation and exploitation working together.

Application currently designed are creating more threads, from multiple runtimes, with specialised use. This creates applications with dynamic workloads.

Listing 6.2 – Example of an Emerging Application using Runtime Stacking

```
1  START Application
2      Worker set A
3      Worker set B
4      A Complete
5      Worker set C
6      C complete
7      B Complete
8  END Application
```

The code presented in Listing 6.2 is a representation of such applications.

From the start of the application to the end, we see multiple worker set being created and destroyed. They are not created and destroyed linearly. Worker set A is the first created and destroyed. Worker set B is created while the A is alive but is destroyed way after A finished. The C worker set is created and destroyed while the B one is alive. If we look at resource usage during all of that, depending on the number of worker in each set, we could have period of time with a lot of threads alive, and some other with a minimal number.

If the resource manager and the hypervisor could communicate, the resource manager could allocate more resource to certain worker sets, or could use the resources for other applications. If in addition the user could give hints in his code, he could for example tell the hypervisor 'this worker set needs at least x threads but no more than y'. And if this information is carried back to the resource manager, then it could use resources more efficiently.

Part III

Conclusion and Future Work

Chapter 7

Conclusion

Numerical simulation is a major asset both for research and industry. It is a catalyst for innovation. However, computing representations of physical phenomenon requires specific techniques and powerful machines. High Performance Computing (HPC) provides specialised solutions to run fast and reliable simulations. Unfortunately, this compute power comes at a cost: architectures are more and more complex and specialised.

Since the creation of computers, the goal as been to improve their capabilities and performances. At first, making processors run faster was the focal point. Then, when limits were reached, the addition of cores on processors became the trend. Now the focus is made on specialised highly parallel hardware. All these evolutions led to complex but powerful heterogeneous machines.

To help harness supercomputers' potential, simulations make use of parallel programming models, each embedding its own runtime. To extract a maximum of performances from the machines, multiple runtimes are often used in the same code, creating runtime stacking situations. However, little research has been conducted on this new problematic. Using and mixing multiple runtimes requires specific techniques and expertise. These techniques, designed to improve applications' performances can lead to situation that are detrimental to them. Runtime stacking is necessary to exploit clusters' architectures to their maximum, but interferences that come from it can hinder global performances. In this context, this thesis proposes a study of runtime stacking techniques and their effects on applications' performances. This study led to the development of tools to assist application developers in detecting resource misuse as well as in optimising runtime behaviours.

7.1 Contributions Summary

Runtime Stacking Taxonomy As runtime stacking emerged as the solution to leverage today supercomputers full performance, we explored how such stacking can happen and behave. Stacking provides a frame to optimise codes, increase the scalability of existing applications and extract the most out of complex architectures. However, it also requires more complex resource management. Based on the bibliography, we define a state-of-the-art taxonomy for runtime stacking in two parts. First, the runtime stacking configurations illustrate how runtime share resources at execution time. Then, runtime stacking categories describe how runtime stacking codes are written. We use both these classifications to determine what kind of optimisations are available for each category of code and to characterise missuses that can happen at execution time.

Algorithms and Tools We then present algorithms that analyse resource usage of an application. These algorithms can be used to determine if any resource has been overloaded or underused at any given time during an application's execution. We design a tool, called the Overseer, that collects information relevant for the algorithms' analysis. It then produces warnings and hints in regard to the resource usage using the algorithms. As we explained, our choices led to a lightweight tool, that can be used anytime without changing the development process of an application (source modification, or compilation chain additions).

Dynamic Resource Management Finally, we presented a second tool, named the Overmind, that dynamically adapts the execution flow placement to optimise applications' performances. Our approach uses code, observations and experiences from the Overseer to consider all threads/processes alive at the same time, and map them efficiently onto the architecture topology. Once again, we designed a lightweight tool that can be used with any application without modification. Experiments show that the overhead of this method is quite low and it is able to dynamically adapt the execution-flow placement to reach better performance on different benchmarks (CORAL) and various architectures (from multicore to manycore).

7.2 Contributions' Perspectives

Larger vision for Overseer and Overmind In the near future, both the Overseer and the Overmind could be improved on. First, they are for the moment focused on compute cores and hyperthreads as resources. Their vision could be broadened by taking more resources into account. Memory and data

placement seems the most beneficial upgrade in the short term. However, in the future, architecture might propose specialised cores, designed to perform some operations better than others. If this is the case, determining which thread is in charge of performing these operations and placing it on these dedicated cores will be the next step.

Analysis and Profiling Tools On another note, we think that continuing development on this tool is important as runtime stacking can have a great impact on applications' performances. We described in each chapter some improvements and future work for both these tools, but the most important one is certainly their integration into larger projects. The Overseer's insight could be coupled to already existing profiling tools. We think that adding an architecture and runtime analysis to this tool would improve the scope of profiling tools. Indeed, runtimes' threads placement on resources could be crucial as workers from a group have a tendency to share memory and communicate with each other. Tools, are designed to pinpoint the scalability bottlenecks like communications and synchronizations, would benefit from the knowledge of which thread belongs to which runtime, or worker set. And conversely, knowing which part of the code was the bottleneck could improve the hints and warnings provided by the Overseer. Indeed, maybe workers from a set were placed on different sockets or NUMA nodes, but didn't need to communicate (or communications were not the bottleneck). In this case, solving the placement 'misuse' would not be critical or needed at all. The analysis could be focused on the bottlenecks and give a certain degree in the warnings with how much the issue slowed the execution.

Dynamic Resource Management Tool Concerning the Overmind, the hypervising tool, we still have a lot of improvements to do. The tool's state presented is still just a proof of concept. We demonstrated that runtime placement and resource usage can be improved on by this type of hypervising software. However, we only implemented one algorithm. This algorithm works with the current architectures and with the group of codes we tested it on. However, we can still need to study code behaviour. We assumed that most current codes would need workers from the same set as close a possible to each other, while in the meantime not overloading resources. This assumption could be false or slightly incorrect for other codes. In this case we would need to change the analysis and placement algorithms to provide more flexibility. For example, some runtimes make use of threads specialised in performing communication operations. Other use helper threads to perform certain operations. The placement of these threads can change application performance. They can be spawned on cores close to the rank performing the calling the communication

or help function. They can also be placed on two hardware threads of the same core. They could even be placed on cores dedicated to helper threads on the node. Other threads could use dedicated cores like the ones spawned by the Operating System for example.

Make Use of External Information Lastly, we thought of an improvement that could be implemented in both the Overmind and the Overseer. We assume that the user can have an idea of the runtime stacking he is using. This information can be used to determine what kind of leverage he can have on his runtimes behaviour. Coupled with the knowledge of which runtime is used and the architecture's topology, we could give basic command line arguments to launch the application and allocate resources. With the information of which runtime is used, the arguments would be more precise, with specific runtime options. Lastly, we assume that the user also knows how he wants his runtime to behave if he is aiming for a non-standard configuration (for example allocating processors for their memory but not using computing resources). We plan on developing a simple language for the user to describe the configuration he wants to achieve. Passing this configuration to our tool would either give him the command line arguments to use in the case of the Overseer, or give more information for the Overmind to find the best configuration. Getting this external type of information could also come from previous executions. Assuming the behaviour will be the same on two consecutive runs, our tools could look at previous logs to optimise resource usage by already knowing which runtimes will be spawned, when, as well as how many workers they will use. This would reduce thread movement as they would be placed on the right cores from the start when possible.

7.3 General Perspectives

One of the goals of this thesis was to point out problems that can arise when mixing multiple parallel programming models. Runtime stacking and resources usage have a great impact on HPC simulations' performances. This impact will likely increase with future HPC machines and codes. Thus, if programming techniques continue to revolve around runtimes and their mixing, runtime stacking needs to be one of the principal focus in HPC performance optimisation. With the increasing complexity of machines' architecture, and the multiplication of runtimes, code development is meant to also become more complex. However, it is not possible for experts in mathematics, physics, biology or any other science to also be computer science and HPC experts. It is necessary to improve development and optimisation techniques to be accessible. Dynamic management of runtimes and resources with third party software is a

Listing 7.1 – Example of an Emerging Application using Runtime Stacking

```
1  START Application
2      Worker set A
3      Worker set B
4      A Complete
5      Worker set C
6      C complete
7      B Complete
8  END Application
```

first step. But other solutions needs to be considered.

Let us first look back at the predicted emerging application's design in Listing 7.1. Worker sets lifespan interweave and the number of worker is varying during the application's execution. This new kind of application bring new problematics. How to allocate resources at application startup? Should we allocate as many resources as possible to optimise the times the maximum of worker are present? In this case, some resources would be idle for an amount of time, wasting compute power and energy. Or should we allocate less resources, and overload them if the number of worker is too large? This solution would have a negative impact on application's performances. To better exploit cluster's resources, a dynamic management of runtimes is necessary. Here are leads that can be explored.

Central Runtime Management One solution could be for runtimes to be managed by themselves instead of needing the addition of another layer of software. A prerequisite would be runtimes discussing, negotiating, sharing and exchanging resources. All runtimes would need to implement a common API to share topologies for example, or to ask for certain number and types of resources. If we want to keep a third party, we could imagine a hub for runtimes to update their current status. Are they doing computations, waiting for resources, waiting for communications, how many and which resources are they using, for how long and so on. New runtimes could then know which resources to use in priority, or start negotiating for resources. These ideas require a different runtime management. Indeed, runtime would need to be able to change parallel sections size dynamically, re-arrange worker groups, or even share workers with each other. The scheduler and OS would also certainly be involved to change cpusets, and allow runtimes more or less resources.

Finer Runtime and Resource Management opportunities In parallel to a central runtime management, new management opportunities can be given to code developers. Indeed, some code parts and algorithms can require a finer management than others. Giving more information to runtimes to manage these parts more precisely could lead to better resource usage. Code developers would however have to adapt to new coding techniques and paradigms. For example, giving a range of needed resources for each parallel section would permit a finer resource management. Instead of creating and needing a fixed number of workers, the parallel section could use between a variable number of threads depending on the other runtimes, and available resources. This solution could be considered for code sections limited in scalability, which would not be efficient with infinite amount of resources. We can also imagine codes using worker threads and also helper threads. This code could ask for one helper thread every x worker thread. The number of worker and helper threads would be determined at execution time depending on available resources.

Using Clusters to their Maximum Potential Usually clusters are not used at 100% of their capabilities. During the execution of a simulation, some resources are idle while other are working. An already existing idea is to implement the same algorithm multiple times, optimised for specific hardware. One copy would be optimised for accelerator type hardware like GPUs, the other for CPUs, another for KNL and so on. Depending on cluster's architecture and other runtimes' needs, the most efficient solution would be determined. With the increasing complexity of architecture and their tendency to evolve towards heterogeneous designs, this type of solution will certainly gain popularity.

Almighty Resource Manager Previous solutions are about adding a layer managing runtimes and resources. An other solution is for actors already present to evolve and ally. We can for example imagine involving the resource manager. Instead of adding a software layer with a hub to share resources, the resource manager could do this work. It is already in charge of configuring and allocating resources for each job running on clusters. An API could be developed for runtimes and resource manager to communicate. Runtimes could make calls to request more resources, or to relinquish them. The resource manager could then, knowing the state of the machine, remap runtimes on resources. This solution has the drawback of needing extensive developments in resource managers, runtimes, and in a shared API. However, its huge benefit is in the centralisation of the solutions.

Optimise Resource Usage with Lightweight Workloads Finally, we spoke a lot of large, time-consuming and resource hungry simulations. However,

HPC clusters are not only used by thousand-cores simulations, they are also exploited by more modest simulations or jobs. Some jobs only need a couple of nodes, or sometimes less than a node to compute. Some executions are only a couple minutes long. Exploiting a cluster's resources to their maximum could also come through optimising small job placement. This would be more efficient as more jobs could run at the same time. It would also be more energy efficient as fewer processors and cores' resources would be wasted. This still comes back to emerging applications using varying numbers of workers and worker set during their execution. Free resources, in low worker count steps could be exploited by small or short jobs waiting for allocation.

Bibliography

[1] J. Jung, W. Nishima, M. Daniels, G. Bascom, C. Kobayashi, A. Adedoyin, M. Wall, A. Lappala, D. Phillips, W. Fischer, *et al.*, "Scaling molecular dynamics beyond 100,000 processor cores for large-scale biophysical simulations", *Journal of computational chemistry*, 2019.

[2] J.-M. Alimi, V. Bouillot, Y. Rasera, V. Reverdy, P.-S. Corasaniti, I. Balmes, S. Requena, X. Delaruelle, and J.-N. Richet, "First-ever full observable universe simulation", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, 2012, p. 73.

[3] D. W. Walker, "Standards for message-passing in a distributed memory environment", Oak Ridge National Lab., TN (United States), Tech. Rep., 1992.

[4] OpenMP Architecture Review Board, *OpenMP application program interface version 5.0*, 2018. [Online]. Available: https://www.openmp.org/specifications/.

[5] J. L. Hennessy and D. A. Patterson, *Computer architecture, fifth edition: a quantitative approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 012383872X, 9780123838728.

[6] C. Carvalho, "The gap between processor and memory speeds", 2002.

[7] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[8] ——, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[9] S. Mittal, "A survey of techniques for dynamic branch prediction", *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, e4666, 2019.

[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967, pp. 483–485.

[11] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castanos, P. Crumley, C. C. Erway, D. Lieber, X. Martorell, J. E. Moreira, *et al.*, "An overview of the bluegene/l system software organization", *Parallel processing letters*, vol. 13, no. 04, pp. 561–574, 2003.

[12] A. Geist and R. Lucas, "Major computer science challenges at exascale", *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 427–436, 2009.

[13] Top500. (). Top500, [Online]. Available: https://www.top500.org/.

[14] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *Linpack users' guide*. SIAM, 1979.

[15] M. E. Conway, "A multiprocessor system design", in *Proceedings of the November 12-14, 1963, fall joint computer conference*, ACM, 1963, pp. 139–146.

[16] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface", *Supercomputer*, vol. 12, pp. 56–68, 1996.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: goals, concept, and design of a next generation mpi implementation", in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2004, pp. 97–104.

[18] M. Team. (1992). Mpich, [Online]. Available: https://www.mpich.org/.

[19] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff, "Mpi at exascale", *Procceedings of SciDAC*, vol. 2, pp. 14–35, 2010.

[20] M. Pérache, H. Jourdren, and R. Namyst, "MPC: a unified parallel runtime for clusters of NUMA machines", in *Proceedings of Euro-Par '08*, Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008, pp. 78–88, ISBN: 978-3-540-85450-0. DOI: 10.1007/978-3-540-85451-7_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_9.

[21] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory", Livermore, CA, Tech. Rep. LLNL-TR-490254, pp. 1–17.

[22] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes", Livermore, CA, Tech. Rep. LLNL-TR-641973, Aug. 2013, pp. 1–9.

[23] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing-to thread or not to thread?", in *2008 IEEE International Conference on Cluster Computing*, IEEE, 2008, pp. 213–222.

[24] A. Denis, J. Jaeger, E. Jeannot, M. Pérache, and H. 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*, vol. 33, no. 6, pp. 1240–1254, 2019.

[25] A. Loussert, B. Welterlen, P. Carribault, J. Jaeger, M. Pérache, and R. Namyst, "Resource-management study in hpc runtime-stacking context", in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2017, pp. 177–184.

[26] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin, "Scalability evaluation of barrier algorithms for openmp", in *International Workshop on OpenMP*, Springer, 2009, pp. 42–52.

[27] P. Carribault, M. Pérache, and H. Jourdren, "Enabling low-overhead hybrid mpi/openmp parallelism with mpc", in *International Workshop on OpenMP*, Springer, 2010, pp. 1–14.

[28] C. Lameter *et al.*, "Numa (non-uniform memory access): an overview.", *Acm queue*, vol. 11, no. 7, p. 40, 2013.

[29] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe", *ACM Sigplan Notices*, vol. 45, no. 6, pp. 376–387, 2010.

[30] P. Carribault, M. Pérache, and H. Jourdren, "Enabling low-overhead hybrid mpi/openmp parallelism with mpc", English, in *Proceedings of IWOMP 2010*, ser. Lecture Notes in Computer Science, vol. 6132, Springer Berlin Heidelberg, 2010, pp. 1–14, ISBN: 978-3-642-13216-2. DOI: 10.1007/978-3-642-13217-9_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13217-9_1.

[31] M. Sabahi, *Getting code ready for parallel execution with intel® parallel composer*, https://software.intel.com/en-us/articles/getting-code-ready-for-parallel-execution-with-intel-parallel-composer, Accessed: 2017-06-12.

[32] A. Marochko, *Tbb 3.0 task scheduler improves composability of tbb based solutions. part 1*, https://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based -solutions-part-1, Accessed: 2017-06-12.

[33] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing", in *Proceedings of HPDC 1999*, IEEE, 1999, pp. 115–124.

[34] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures", *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[35] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: a supervised approach", *The International Journal of High Performance Computing Applications*, vol. 28, no. 3, pp. 285–300, 2014.

[36] R. Bagrodia, R. Meyer, M. Takai, Y.-a. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: a parallel simulation environment for complex systems", *Computer*, vol. 31, no. 10, pp. 77–85, 1998.

[37] G. developers. (). Gdb, [Online]. Available: https://www.gnu.org/software/gdb/.

[38] inside HPC. (2012). How allinea helps scientists run their applications faster and better, [Online]. Available: https://www.gnu.org/software/gdb://insidehpc.com/2012/12/how-allinea-helps-scientists-run-their-applications-faster-and-better/.

[39] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "Marmot: an mpi analysis and checking tool", in *Advances in Parallel Computing*, vol. 13, Elsevier, 2004, pp. 493–500.

[40] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "Mpi runtime error detection with must: advances in deadlock detection", *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[41] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture", *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[42] S. S. Shende and A. D. Malony, "The tau parallel performance system", *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[43] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs", *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[44] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, *et al.*, "Score-p: a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir", in *Tools for High Performance Computing 2011*, Springer, 2012, pp. 79–91.

[45] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: visualization and analysis of mpi resources", 1996.

[46] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: a tool to visualize and analyze parallel code", in *Proceedings of WoTUG-18: transputer and occam developments*, IOS Press, vol. 44, 1995, pp. 17–31.

[47] S. Picault and P. Mathieu, "An interaction-oriented model for multi-scale simulation", in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[48] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c", in *Tools for High Performance Computing 2009*, Springer, 2010, pp. 157–173.

[49] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems", *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[50] L. Smith and M. Bull, "Development of mixed mode mpi/openmp applications", *Scientific Programming*, vol. 9, no. 2-3, pp. 83–98, 2001.

[51] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes", in *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, IEEE, 2009, pp. 427–436.

[52] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++", in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IEEE, 2014, pp. 1105–1114.

[53] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: shmem for the pgas community", in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ACM, 2010, p. 2.

[54] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: exploiting heterogeneity to enhance scalability", *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[55] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss", *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[56] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures", *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[57] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures", *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, 2 Feb. 2011. DOI: 10.1002/cpe.1631. [Online]. Available: http://hal.inria.fr/inria-00550877.

[58] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing", in *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*, IEEE, 1999, pp. 115–124.

[59] G. R. Vallée and D. Bernholdt, "Improving support of mpi+ openmp applications", 2018.

[60] S. K. Gutiérrez, K. Davis, D. C. Arnold, R. S. Baker, R. W. Robey, P. McCormick, D. Holladay, J. A. Dahl, R. J. Zerr, F. Weik, *et al.*, "Accommodating thread-level heterogeneity in coupled parallel applications", in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2017, pp. 469–478.

[61] B. Goglin, "Exposing the locality of heterogeneous memory architectures to hpc applications", in *Proceedings of the Second International Symposium on Memory Systems*, ACM, 2016, pp. 30–39.

[62] E. A. León, "Mpibind: a memory-centric affinity algorithm for hybrid applications", in *Proceedings of the International Symposium on Memory Systems*, ACM, 2017, pp. 262–264.

[63] A. Vega, A. Buyuktosunoglu, and P. Bose, "Smt-centric power-aware thread placement in chip multiprocessors", in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE, 2013, pp. 167–176.

[64] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors", in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, IEEE, 2010, pp. 491–496.

[65] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on numa systems: asymmetry matters", in *2015 USENIX Annual Technical Conference*, 2015, pp. 277–289.

[66] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "Hwloc: a generic framework for managing hardware affinities in hpc applications", in *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 180–186, ISBN: 978-0-7695-3939-3. DOI: 10.1109/PDP.2010.67. [Online]. Available: http://dx.doi.org/10.1109/PDP.2010.67.