Co-scheduling for large-scale applications : memory and resilience

Loïc Pottier


Numéro National de Thèse : 2018LYSEN039

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

*École Doctorale N°*512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 18/09/2018, par :

Loïc POTTIER

Co-scheduling for large-scale applications: memory and resilience

Ordonnancement concurrent d'applications à grande échelle: mémoire et résilience

Devant le jury composé de :

Anne	BENOIT	Maître de Conférences, ENS de Lyon	*Directrice de thèse*
Élisabeth	BRUNET	Maître de Conférences, Télécom SudParis	*Examinatrice*
Michel	DAYDÉ	Professeur, IRIT, Toulouse	*Examinateur*
Emmanuel	JEANNOT	Directeur de recherche, Inria, Bordeaux	*Rapporteur*
Pierre	MANNEBACK	Professeur, Polytech-Mons (Belgique)	*Rapporteur*
Yves	ROBERT	Professeur, ENS de Lyon	*Co-encadrant de thèse*

Introduction

In 2005, computational science has been established as the third *pillar* of science by the President's Information Technology Advisory Committee [100]. Computational science has become a critical tool for a better understanding of major scientific challenges in numerous areas, such as weather forecasting, climate prediction, artificial intelligence or nuclear programs. The interest of computational science is mainly driven by the processing capabilities of supercomputers, or high performance computing (HPC) systems, running large-scale simulations. The processing capability of a supercomputer is defined as the number of floating point operations (FLOP) it can achieve in one second. In the example of weather forecasting, the higher the processing capability, the more accurate the model predictions. Developing more and more powerful HPC systems is an active research area [2, 37]. The most powerful supercomputers are currently running at Petascale (10^{15} floating point operation per second) [44]. In parallel, several governments or institutions are now targeting the *Exascale* (i.e., 10^{18} floating point operation per second!), and the America's first Exascale supercomputer is expected for 2021. Recently, in January 2018, the European Commission unveiled a plan to invest one billion euros[1] into a world-class European supercomputers and into research for future Exascale systems. Future Exascale systems will be massively parallel, composed of hundreds of thousands processing units [2, 37, 99]. Such systems raise a lot of challenging problems about their feasibility; hence new scientific breakthroughs are needed, both on the hardware side (power efficient and reliable architectures) and on the software side (scalable algorithms and software systems).

Two studies [2, 99] pointed out major issues on the road of the Exascale computing, as diverse as: power efficiency, scalable algorithms and software, resilience and correctness, using emerging architectures and massive concurrency. This thesis deals with two prominent problems in this list, namely, concurrency and resilience at scale. In the last part, we also start to explore the problem of scheduling workflows on emerging architectures, like the Xeon Phi Knights Landing.

A classic scheduling strategy for HPC platforms is to execute each application on a dedicated node. With the recent advent of many-core architectures such as chip multiprocessors (CMP), the number of processing units by node is constantly increasing. Future Exascale platforms are expected to exhibit a thousand times more concurrency than current Petascale systems [2]. Unless the application that runs alone on a dedicated node is perfectly parallel, the efficiency of such massively concurrent nodes will decrease. In 1967, Amdahl established a law to model the execution time of parallel applications [3]. According to Amdahl's law, an application will execute, on p processors, in time

$$s \times t_{seq} + (1 - s)\frac{t_{seq}}{p},$$

where s is the fraction of sequential time and t_{seq} is the sequential execution time. A perfectly parallel application has a sequential fraction s equal to zero; hence a perfectly parallel application has an execution time t_{seq}/p. In practice, because of the overhead due to communications and to the inherently

[1] http://europa.eu/rapid/press-release_IP-18-64_en.htm

sequential fraction of the application s, the parallel execution time is larger than t_{seq}/p. According many studies [2, 99], compute nodes at Exascale will be massively parallel, in other words, p will be large. Under this assumption, the execution time will be bounded by the sequential fraction s (and also by communication overheads, not taken into account by Amdahl's law). Several solutions are available: (i) develop scalable algorithms in order to reduce the sequential fraction, or (ii) use a *co-scheduling* approach to improve node efficiency. In this manuscript, we focus on the second solution. The main idea of *co-scheduling* is to execute several applications concurrently, rather than in sequence, with the objective to increase the node efficiency. When multiple application are concurrently scheduled, or *co-scheduled*, onto a platform, they will compete for shared resources, as cache memories or network and I/O links, and create interferences, or *co-run degradations*. The main difficulty of co-scheduling is to decide which applications to execute concurrently in order to reduce potential interferences and how many resources should be assigned to each of them. We investigate this challenging problem, focusing on interferences in the last-level cache (LLC), in Chapters 2 and 3.

While massive concurrency is a major challenge for Exascale, another critical challenge is the reliability of future Exascale platforms. In February 2014, the Advanced Scientific Computing Advisory Committee (ASCAC) established a list of ten research challenges [99], resilience and correctness is one them. The resilience is defined by the ASCAC as "ensuring correct scientific computation in face of faults, reproducibility and algorithm verification challenges". The mean time between failures (MTBF) of the upcoming generations of Exascale systems is expected to be a major issue [26, 27]. Let μ_{ind} be the MTBF of an individual processor. Then, the MTBF of a platform with p identical processors is equal to [58, Proposition 1.2]:

$$\mu_p = \frac{\mu_{ind}}{p}.$$

We can clearly observe how the resilience problem is directly linked to the increasing level of concurrency (when p increases). In Chapter 4, we study how resilience can impact co-scheduling performance and how faults can be taken into account when we want to minimize the maximum completion time of several co-scheduled applications.

Future Exascale systems will probably rely on new massively parallel architectures, such as many-core systems. Recently, many TOP500 supercomputers [44] use many-core architectures to increase their processing capabilities, such as the Intel Knights Landing (KNL). Some of these new architectures exhibit also a new high-bandwidth on-package memory, and this new memory adds a new level in the memory hierarchy. To exploit at their full potential the future Exascale platforms, building performance models taking into account these new memories is essential. We further investigate this topic in Chapter 5.

The rest of the thesis is organized as follows. In Chapter 1, we thoroughly review the context of this thesis from parallel architectures to scheduling models, with the different problematics and contributions associated. In Chapter 2, we start the study of co-scheduling applications sharing a last-level cache. In Chapter 3, we assess the interest of cache partitioning when co-scheduling HPC workloads, through an experimental campaign on a multiprocessor cache-partitioned system. We continue to explore co-scheduling problems in Chapter 4, where we focus on co-scheduling algorithms in a failure-prone context. Indeed, failures can create severe imbalanced scheduled. By redistributing processors, we show how to minimize the execution time of a given co-schedule. Finally, in Chapter 5, we are interested into workflow scheduling and memory management on new deep-memory many-core architectures. The main contributions of each chapter are summarized below.

Chapter 1: Context and contributions

In this preliminary chapter, we introduce the global context of this thesis and we detail each contribution. Parallel architectures, at the core of actual and future supercomputers, exhibit an increasing number of processing units (or cores). HPC applications are expected to take advantage of that amount of available concurrency. Such applications can easily be represented as a task graph [39], also called a workflow, where each task represents a simple computation, as for instance the multiplication of two matrix tiles [22]. The programmers write the application and then it is the role of the scheduler to optimize the execution of this application on a given architecture by assigning tasks to cores. With the massive concurrency offered by several recent parallel architectures [31, 35, 61], multiple tasks are likely to run concurrently on these platforms. The idea behind co-scheduling is to concurrently execute applications rather than in sequence, and to use the whole platform for each task. But, in these recent parallel architectures, some functionalities, like caches, memory controllers or buses, are shared between compute cores. This may lead to performance degradation when multiple tasks compete for these shared resources, these potential contention must be taken into account to obtain good co-scheduling performance.

Chapter 2: Co-scheduling applications on cache-partitioned systems [W1, J1]

In this chapter, we study the scheduling problem of minimizing the completion time of several concurrent applications running on cache-partitioned architecture. Cache-partitioned architectures allow subsections of the shared last-level cache (LLC) to be exclusively reserved for some applications. This technique dramatically limits interactions between applications that are concurrently executing on a multi-core machine. Consider n applications that execute concurrently, with the objective to minimize the makespan, defined as the maximum completion time of the n applications. Key scheduling questions are: (i) which proportion of cache and (ii) how many processors should be given to each application? In this chapter, we provide answers to (i) and (ii) for Amdahl applications. Even though the problem is shown to be NP-complete, we give key elements to determine the subset of applications that should share the LLC (while remaining ones only use their smaller private cache). Building upon these results, we design efficient heuristics for Amdahl applications. Extensive simulations demonstrate the usefulness of co-scheduling when our efficient cache partitioning strategies are deployed.

Chapter 3: Co-scheduling HPC workloads on cache-partitioned CMP platforms [C3]

Based on the results obtained in Chapter 2, we pursue the study of co-scheduling algorithms with cache partitioning techniques but, this time, using a real cache-partitioned multiprocessor to assess the interest of cache partitioning on such platforms. In this chapter, we focus on the interferences in the last level of cache (LLC) and use the *Cache Allocation Technology* (CAT) recently provided by Intel to partition the LLC and give each co-scheduled application their own cache area. We consider m iterative HPC applications running concurrently, and answer the following questions: (i) how to precisely model the behavior of these applications on the cache partitioned platform? and (ii) how many cores and cache fractions should be assigned to each application to maximize the platform efficiency? Here, platform efficiency is defined as maximizing the performance either globally, or as guaranteeing a fixed ratio of iterations per second for each application. Through extensive experiments using CAT, we demonstrate the impact of cache partitioning when multiple HPC application are co-scheduled onto CMP platforms.

Chapter 4: Resilient co-scheduling of malleable applications [C1, B1, J2]

After focusing on memory in Chapters 2 and 3, we now discuss how a failure-prone framework impacts co-scheduling performance. Indeed, the benefits of co-scheduling several applications have been demonstrated in a fault-free context, both in terms of performance and energy savings. However, large-scale computer systems are confronted to frequent failures, and resilience techniques must be employed for large applications to execute efficiently. Indeed, failures may create severe imbalance between applications, and significantly degrade performance. In this chapter, we aim at minimizing the expected completion time of a set of co-scheduled applications. We propose to redistribute the resources assigned to each application upon the striking of failures, and upon the completion of some applications, in order to achieve this goal. First, we introduce a formal model and establish complexity results. The problem is NP-complete for malleable applications, even in a fault-free context. Therefore, we design polynomial-time heuristics that perform redistributions and account for processor failures. A fault simulator is used to perform extensive simulations that demonstrate the usefulness of redistribution and the performance of the proposed heuristics.

Chapter 5: A performance model to execute workflows on high-bandwidth-memory architectures [C2]

This chapter presents a realistic performance model to execute scientific workflows on high-bandwidth-memory architectures such as the Intel Knights Landing. We provide a detailed analysis of the execution time on such platforms, taking into account transfers from both fast and slow memory and their overlap with computations. We discuss several scheduling and mapping strategies: not only tasks must be assigned to computing resources, but also one has to decide which fraction of input and output data will reside in fast memory and which will have to stay in slow memory. We use extensive simulations to assess the impact of the mapping strategies on performance. We also conduct experiments for a simple 1D Gauss-Seidel kernel, which assess the accuracy of the model and further demonstrate the importance of a tuned memory management. Our model and results lay the foundations for further studies and experiments on dual-memory systems.

Contents

French summary

En 2005, les sciences numériques ont été définies comme le troisième pilier des sciences par le « President's Information Technology Advisory Committee » [100]. Les sciences numériques sont devenues un outil essentiel pour une meilleure compréhension de nombreux défis scientifiques majeurs tels que, la météorologie, la prédiction climatique, l'intelligence artificielle, ou les programmes nucléaires. L'intérêt des sciences numériques réside principalement dans la puissance de calcul des supercalculateurs, ou *high performance computing* (HPC), exécutant des simulations à grande échelle. La puissance de calcul d'un supercalculateur est définie comme: le nombre d'opérations en virgule flottante (FLOP) qu'il peut effectuer en une seconde. En reprenant l'exemple de la prédictions météorologique, plus un supercalculateur est puissant, plus les prédictions du modèle météorologique seront précises. Développer des supercalculateurs toujours plus puissants est une thématique de recherche très active [2, 37]. À l'heure actuelle, les plus puissants supercalculateurs ont une puissance *petaflopique* (10^{15} opérations en virgule flottante par seconde) [44]. En parallèle, plusieurs gouvernements et institutions de recherche commencent à planifier les futurs supercalculateurs *exaflopiques* (i.e., 10^{18} opérations en virgule flottante par seconde!), le premier supercalculateur américain est prévu pour l'horizon 2021. Récemment, en janvier 2018, la Commission Européenne a révélé un plan d'investissement d'un milliard d'euros[2] pour financer des supercalculateurs européens de classe mondiale et pour financer la recherche sur un futur système exaflopique. Les futurs supercalculateurs de classe exaflopique seront massivement parallèle, composés de centaines de milliers d'unités de traitements [2, 37, 99]. De tels systèmes soulèvent de nombreux défis de faisabilité; cela requiert de nouveaux progrès scientifiques, au niveau matériel (architectures efficaces énergétiquement et tolérantes aux pannes) ainsi qu'au niveau logiciel (algorithmes passant à l'échelle et systèmes d'exploitations adaptés).

Deux études [2, 99] dressent une liste des problèmes les plus importants sur la route des calculateurs exaflopiques, aussi divers que: efficacité énergique, algorithmes et systèmes passant à l'échelle, résilience, utilisation de nouvelles architectures, et une concurrence massive. Cette thèse traite de deux problèmes majeurs présents dans cette liste, la concurrence et la résilience à l'échelle. Dans la dernière partie de cette thèse, nous explorons également le problème de l'ordonnancement d'un graphe de tâches sur des nouvelles architectures, comme les Xeon Phi Knights Landing.

Une stratégie classique d'ordonnancement pour les systèmes HPC est d'exécuter chaque application sur un nœud dédié. Avec le récent engouement pour les architectures massivement parallèles, type *many-core*, le nombre d'unités de traitement ne cesse d'augmenter. Les futurs systèmes exaflopiques proposeront un millier de fois plus de concurrence que les systèmes petaflopiques actuels [2]. À moins que l'application s'exécutant seule sur le nœud de calcul ne soit parfaitement parallèle, l'efficacité d'un nœud massivement parallèle va décroître. En 1967, Amdahl propose une loi pour modéliser le temps d'exécution d'une application parallèle [3]. Selon la loi d'Amdahl, une application qui s'exécutera sur

[2] http://europa.eu/rapid/press-release_IP-18-64_en.htm

p processeurs mettra un temps

$$s\, t_{seq} + (1 - s)\frac{t_{seq}}{p},$$

où s est la fraction de temps séquentielle et t_{seq} est le temps d'exécution séquentiel. Une application parfaitement parallèle a une fraction séquentielle s égale à zéro; par conséquent une application parfaitement parallèle a un temps d'exécution t_{seq}/p. En pratique, à cause du surcoût dû aux communications et à la fraction séquentielle intrinsèque de l'application s, le temps d'exécution parallèle réel est plus grand que t_{seq}/p. Selon plusieurs études [2, 99], les nœuds de calculs exaflopiques seront massivement parallèle, en d'autres termes, p va croître. Selon cette hypothèse, le temps d'exécution sera borné par la fraction séquentielle s (ainsi que par les surcoûts de communications non pris en compte par la loi d'Amdahl). Plusieurs solutions sont possibles: (i) concevoir des algorithmes passant à l'échelle pour réduire la fraction séquentielle s, ou (ii) utiliser l'approche du *co-ordonnancement* pour améliorer l'efficacité du nœud de calcul. Dans ce manuscrit, nous nous focalisons sur la seconde solution. L'idée principale du *co-ordonnancement* est d'exécuter plusieurs applications de manière concurrente, plutôt que de manière séquentielle, avec l'objectif d'augmenter l'efficacité du nœud de calcul. Quand plusieurs applications sont ordonnancées de manière concurrente, ou *co-ordonnancées*, sur un nœud de calcul, elle vont se disputer les ressources partagées, comme les antémémoires (mémoires caches) ou le réseau et les systèmes d'entrées/sorties, et vont créer des interférences, aussi appelé *co-run degradations*. La principale difficulté du *co-ordonnancement* est de décider quelle application exécuter de manière concurrente avec quelle autre, avec l'objectif de réduire les potentielles interférences, et combien de ressources doivent être alloué à chaque application. Nous étudions ce problème, en nous focalisant sur les interférences dans le dernier niveau de mémoire cache, dans les chapitres 2 et 3.

Tandis que la concurrence massive est un défi majeur pour les plates-formes exaflopiques, un autre défi est celui de la résilience sur de telles plates-formes. En février 2014, l'Advanced Scientific Computing Advisory Committee (ASCAC) a établi une liste de dix problématiques de recherche, la résilience est l'une d'entre elles. La résilience est définie par l'ASCAC comme « ensuring correct scientific computation in face of faults, reproducibility and algorithm verification challenges ». Le temps moyen entre chaque panne (MTBF) des prochaines générations de plates-formes exaflopiques deviendra un problème majeur [26, 27]. Soit μ_{ind} le MTBF d'un seul processeur. Alors, le MTBF d'une plate-forme avec p processeurs identiques est égal à [58, Proposition 1.2]:

$$\mu_p = \frac{\mu_{ind}}{p}.$$

Nous pouvons facilement observer comment le problème de la résilience est directement lié à la concurrence massive qui ne cesse d'augmenter (p augmente). Dans le chapitre 4, nous étudions comment la résilience peut impacter les performances des ordonnancements concurrents et comment les pannes peuvent être prise en compte pour minimiser le temps maximum de terminaison des applications quand plusieurs d'entre elles sont co-ordonnancées.

Les futures plates-formes exaflopiques, seront probablement basées sur des architectures massivement parallèles, comme les systèmes many-core. Récemment, beaucoup de supercalculateurs du TOP500 [44] utilisent les architectures many-core pour augmenter leur puissance de calcul, comme par exemple les architectures Intel Knights Landing (KNL). Plusieurs de ces nouvelles architectures offrent un nouveau niveau dans la hiérarchie mémoire avec une mémoire haute performance. Pour exploiter les futurs systèmes exaflopiques, construire des modèles de performance qui prennent en compte ces nouvelles mémoires est crucial. Nous approfondissons cette thématique dans le chapitre 5.

Le reste de cette thèse est organisée de la façon suivante. Dans le chapitre 1, nous passons en revue le contexte autour de cette thèse des architectures parallèles aux modèles d'ordonnancements, avec les

différents problématiques ainsi que les contributions associées. Dans le chapitre 2, nous commençons l'étude du co-ordonnancement d'applications partageant le dernier niveau de cache. Dans le chapitre 3, nous évaluons l'intérêt du partitionnement de cache quand on co-ordonnance plusieurs applications HPC, grâce à une campagne d'expérimentations sur une plate-forme multi-processeurs permettant le partitionnement de cache. Nous continuons d'explorer les problèmes de co-ordonnancement dans le chapitre 4. Dans ce chapitre, nous nous focalisons sur les algorithmes de co-ordonnancement dans un contexte résilient, en effet les pannes peuvent créer des ordonnancements fortement déséquilibrés. Avec des redistributions de processeurs, nous montrons comment minimiser le temps d'exécution d'un co-ordonnancement donné. Dans le chapitre 5, nous nous intéressons à l'ordonnancement d'un graphe de tâches, représentant une application complexe, ainsi qu'à la gestion mémoire sur ces nouvelles architectures many-core avec une hiérarchie mémoire profonde. Les contributions principales de chaque chapitre sont résumées ci-dessous.

Chapitre 1: Contexte et contributions

Dans ce chapitre introductif, nous détaillons le contexte global de cette thèse ainsi que chaque contribution. Les architectures parallèles, au cœur des supercalculateurs actuels et futurs, présentent un nombre croissant d'unités de traitement (ou cœurs de calcul). Les applications HPC sont censées profiter de cette quantité de concurrence disponible. De telles applications peuvent facilement être représentées sous la forme d'un graphe de tâches [39], aussi appelé *workflow*, où chaque tâche représente un calcul simple, comme la multiplication de deux blocs d'une matrice par exemple [22]. Les programmeurs écrivent l'application et c'est ensuite le rôle de l'ordonnanceur d'optimiser l'exécution de cette application pour une architecture donnée en assignant des tâches aux cœurs de calcul. Avec la concurrence massive offerte par plusieurs architectures parallèles récentes [31, 35, 61], plusieurs tâches sont susceptibles de s'exécuter simultanément sur ces plates-formes. L'idée derrière le co-ordonnancement est d'exécuter simultanément des applications plutôt que de les exécuter les unes après les autres en utilisant la plate-forme entière pour chaque tâche. Mais, dans ces architectures parallèles récentes, certaines fonctionnalités, comme les caches, les contrôleurs mémoire ou les bus sont partagées entre les cœurs de calcul. Cela entraînera une dégradation des performances lorsque plusieurs tâches seront en concurrence pour ces ressources partagées. Ces conflits potentiels doivent être pris en compte pour obtenir de bonnes performances de co-ordonnancement.

Chapitre 2: Co-ordonnancement d'applications sur systèmes à partitionnement de cache [W1, J1]

Dans le premier chapitre de cette thèse, nous étudions le problème d'ordonnancement consistant à minimiser le temps de terminaison de plusieurs applications s'exécutant sur une architecture à partitionnement de cache. Les architectures à partitionnement de cache permettent d'allouer des portions du dernier niveau de cache (LLC) exclusivement réservées à certaines applications. Cette technique permet de réduire drastiquement les interactions entre applications qui sont exécutées simultanément sur une machine multi-cœurs. Considérons n applications exécutées simultanément avec l'objectif de minimiser le *makespan*, défini comme le maximum des temps de complétions parmi les n applications. Les problèmes d'ordonnancement sont les suivants: (i) quelle proportion de cache et (ii) combien de processeurs doivent être alloués à chaque application? Ici, nous assignons des nombres de processeurs rationnels pour chaque application, pour qu'ils puissent être partagés parmi les applications grâce au *multi-threading*. Dans ce chapitre, nous fournissons des réponses aux questions (i) et (ii) pour des applications parfaitement parallèles. Malgré cela, le problème est prouvé être NP-complet, et nous donnons des éléments clés pour déterminer le sous-ensemble des applications qui doivent partager le

dernier niveau de cache (tandis que les autres utilisent seulement leur petit cache privé). Basé sur ces résultats, nous développons des heuristiques efficaces pour des profils d'applications généraux. Un ensemble complet de simulations démontre l'utilité de l'ordonnancement concurrent quand les techniques de partitionnement de cache sont mises en place.

Chapitre 3: Co-ordonnancement d'applications sur des systèmes multi-processeurs à partitionnement de cache [R5]

Basé sur les résultats obtenus dans le chapitre 2, nous poursuivons notre étude des algorithmes de co-ordonnancement avec partitionnement de cache mais, cette fois, en utilisant un système multi-processeurs récent permettant le partitionnement de cache pour démontrer l'intérêt du partitionnement de cache sur de tels systèmes. Avec l'avènement récent des architecture many-core comme par exemple les systèmes multi-processeurs (CMP), le nombre d'unités de traitement communiquant avec une mémoire globale partagée augmente constamment. Les techniques de co-ordonnancement sont utilisées pour améliorer le débit des applications sur de telles architectures, mais partager les ressources génère souvent des interférences importantes. Dans ce chapitre, nous nous focalisons sur les interférences dans le dernier niveau de cache et nous utilisons une technologie appelée *Cache Allocation Technology* (CAT), récemment mise à disposition par Intel, pour partitionner le dernier niveau de cache (LLC) et donner à chaque application co-ordonnancée sa propre zone dans le cache. Nous considérons m applications HPC itératives s'exécutant de manière concurrente et nous répondons aux questions suivantes: (i) comment modéliser précisément le comportement de ces applications sur des architectures supportant le partitionnement de cache? et (ii) combien de cœurs et de fractions de cache doivent être assignés pour maximiser l'efficacité de la plate-forme? L'efficacité de la plate-forme est définie comme le fait de maximiser la performance soit globalement, soit en garantissant un ratio fixe d'itérations pour chaque application. Grâce à de nombreuses expériences utilisant la technologie CAT, nous démontrons l'impact du partitionnement de cache quand plusieurs applications HPC sont co-ordonnancées sur des plates-formes multi-processeurs.

Chapitre 4: Co-ordonnancement d'applications malléables dans un contexte résilient [C1, B1, J2]

Après s'être focalisés sur des aspects mémoires dans les chapitres 2 et 3, nous étudions comment une plate-forme sujette aux pannes peut affecter les performances du co-ordonnancement. En effet, les bénéfices de l'ordonnancement concurrent de plusieurs applications ont été démontrés dans un contexte sans fautes, à la fois en terme de performance et de consommation énergétique. Cependant, les plates-formes distribuées à grande échelle sont fréquemment confrontées à des pannes, et des techniques de résilience doivent être employées. En effet, les pannes peuvent créer des déséquilibres importants entre applications et ainsi dégrader les performances. Dans cet article, nous proposons de redistribuer les ressources allouées à chaque application à chaque fois qu'une faute survient, et quand se termine l'exécution des premières applications, dans le but de minimiser le temps de complétion d'un ensemble de tâches concurrentes. Dans un premier temps, nous introduisons le modèle formel et nous présentons des résultats de complexité. Quand aucune redistribution n'est permise, nous pouvons minimiser l'espérance du temps de complétion en temps polynomial, tandis que le problème devient NP-complet lorsque les redistributions sont permises, même dans un contexte sans fautes. Par conséquent, nous proposons des heuristiques polynomiales effectuant des redistributions, et prenant en compte les pannes des processeurs. Un simulateur de fautes est utilisé pour réaliser un nombre important de simulations qui démontrent l'utilité de la redistribution, ainsi que les performances des heuristiques proposées.

Chapitre 5: Modèle de performance pour exécuter des graphes de tâches sur des architectures à mémoire haute performance [C2]

Ce chapitre présente un modèle de performance réaliste pour exécuter des graphes de tâches scientifiques sur des architectures ayant des mémoires à bande passante élevée, comme par exemple Intel Knights Landing. Nous fournissons une analyse détaillée du temps d'exécution sur ces plates-formes, en tenant compte des transferts depuis deux mémoires (rapide et lente), et leur recouvrement avec les calculs. Nous introduisons plusieurs stratégies d'ordonnancement et de placement mémoire: non seulement les tâches doivent être assignées aux ressources de calcul, mais il faut aussi décider quelle fraction des données d'entrée et de sortie va résider en mémoire rapide, alors que le reste sera en mémoire lente. Des simulations approfondies nous permettent d'évaluer l'impact des stratégies de placement sur la performance. Nous menons également des expériences réelles pour un noyau de Gauss-Seidel 1D simple, afin d'évaluer la précision du modèle. Nous démontrons ainsi l'importance d'une gestion fine de la mémoire sur les systèmes avec double mémoire.

Conclusion

Dans cette thèse, nous avons étudié deux problèmes difficiles, à savoir, la concurrence et la résilience, qui doivent être étudiés pour les futures plates-formes exaflopiques. Dans un premier temps, sur l'aspect concurrent, nous avons étudié le problème de réduction des interférences parmi des applications exécutées de manière concurrente qui partagent le même dernier niveau de cache. Basé sur un modèle de performance détaillé, nous avons évalué la complexité du problème et nous avons conçu des heuristiques efficaces. Nous avons également évalué l'intérêt des techniques de partitionnement de cache sur une plate-forme multi-processeurs existante le supportant. Dans un second temps, nous avons construit un modèle, établi la complexité du problème et conçu des heuristiques efficaces pour s'attaquer au problème du co-ordonnancement d'applications sur une plate-forme pouvant subir des pannes. Après s'être focalisés sur les techniques de co-ordonnancement, nous avons commencé à étudier le problème d'ordonnancement d'un graphe de tâches scientifique sur des architectures émergentes (tels que les many-core) fournissant un nouveau niveau dans la hiérarchie mémoire. Avec le développement des technologies many-core dans le calcul haute performance, ce sujet de recherche semble très prometteur.

Le travail effectué dans cette thèse peut être poursuivi dans plusieurs directions, nous discutons ici des perspectives possibles.

Perspectives et travaux futurs.

Tout au long de cette thèse, à la fin de chaque chapitre, nous avons indiqué plusieurs futures directions de recherches intéressantes. Nous présentons ici quelques conseils pour d'autres directions de recherche prometteuses.

Nous avons étudié le problème du co-ordonnancement en nous concentrant sur deux aspects, à savoir la résilience et les interférences de cache. Du côté du cache, une perspective à court terme consiste à étendre notre analyse expérimentale à d'autres applications et à des plates-formes supportant le partitionnement de cache, afin de mieux étudier les gains potentiels du partitionnement de cache sur les applications HPC. Du coté des perspectives à long terme, une première possibilité intéressante consiste à étendre notre analyse aux plates-formes supportant le partitionnement de bande passante, une fonctionnalité récemment fournie par Intel. Dans les chapitres 2 et 3, nous avons seulement utilisé des techniques de partitionnement de cache pour réduire les interférences, mais une partie non négligeable des interférences se produit dans le bus partagé entre la mémoire principale et le cache. Une seconde perspective serait de généraliser les expériences aux multiprocesseurs et étudier s'il y a un avantage à déplacer des applications d'un processeur à un autre, afin d'éviter la co-location de plusieurs applications *cache-intensive* sur le même processeur. Une troisième perspective serait de trouver une loi plus appropriée pour modéliser les défauts de caches pour les applications HPC. Dans les chapitres 2 et 3, nous avons utilisé la *Power Law* pour modéliser le comportement des défauts de cache. Cette loi nous donne une estimation du nombre de défauts de cache en fonction d'une taille de cache donnée, mais nous avons montré, expérimentalement, que cette loi ne convient pas pour modéliser des applications HPC de type *memory-intensive*. Il serait intéressant de valider un nouveau modèle pour modéliser les défauts de cache.

Du côté de la résilience, que nous avons exploré dans le chapitre 4, plusieurs directions intéressantes peuvent être envisagées. Le premier est d'étendre notre travail aux erreurs silencieuses (*silent errors*) en ajoutant des mécanismes de vérification pour détecter de telles erreurs, et d'étudier le problème d'ordonnancement avec plusieurs groupes d'applications (*pack*) au lieu d'un seul. La deuxième direction est d'étendre notre analyse théorique aux problèmes d'ordonnancement sans connaissance préalable des applications (*online scheduling*) dans un contexte sujet aux pannes.

Enfin, dans la dernière partie de cette thèse, nous avons initié une étude sur le problème de l'ordonnancement de graphes de tâches sur des architectures multi-cœurs présentant des systèmes à double mémoire. Nous avons commencé par étudier les approches d'ordonnancement classiques, mais ces architectures many-core offrent souvent une concurrence massive; par conséquent, elle sont bien adaptée aux techniques de co-ordonnancement. Une orientation de recherche prometteuse consisterait à appliquer notre modèle de co-ordonnancement, basé sur le partitionnement du cache, à ces systèmes à double mémoire massivement parallèles. En effet, nous pouvons considérer la mémoire rapide comme un cache, et utiliser les schémas de partitionnement du cache que nous avons développés précédemment sur cette mémoire rapide. Et, symétriquement aux plates-formes à partition de bande passante discutées ci-dessus, nous pouvons envisager de partitionner la mémoire rapide et la bande passante entre toutes les applications co-ordonnancées, afin d'optimiser l'efficacité globale de la plate-forme.

Chapter 1

Context and contributions

1.1 Context

In a near future, the massive concurrency of parallel architectures in HPC compute nodes is expected to be a critical problem [99]. Parallel architectures, at the core of actual and future supercomputers, exhibit an increasing number of processing units (or cores). HPC applications are expected to take advantage of that amount of available concurrency. Such applications can easily be represented as a task graph [39], also called a workflow, where each task represents a simple computation, like the multiplication of two matrix tiles for example [22]. This paradigm is widely used by many popular task graph schedulers [8, 21, 22, 49], and the task-based approach is also the core of OpenMP 4.0 [92]. This approach has the advantage to abstract the implementation of applications from their execution on parallel architectures. The programmers write the application and then it is the role of the scheduler to optimize the execution of this application on a given architecture by assigning tasks to cores. In this chapter, we use the terms *application* and *task*. When we study co-scheduling problems in Chapters 2 to 4, we focus on a coarse-grain approach, therefore we consider several parallel applications that obey to a given scalability law, such as Amdahl's law [3]. We also adopt, in the last chapter, a finer-grained approach and we study the scheduling problem of one application composed of multiple sequential tasks.

With the massive concurrency offered by several recent parallel architectures [31, 35, 61], multiple tasks are likely to run concurrently on these platforms. Consider for instance the Gyoukou ZettaScaler supercomputer, currently ranked #4 in the TOP500 benchmark [44]: it uses PEZY-SC2, a 2048-core processor chip [31], as emphasized in the introduction, with so many cores at disposal that few applications can efficiently be deployed on the entire computing platform. The idea behind co-scheduling is to concurrently execute applications rather than in sequence and using the whole platform for each application. But, in these recent parallel architectures, some functionalities, such as caches, memory controllers or buses, are shared between compute cores. For example, the PEZY-SC2 platform mentioned earlier is a many-core system where 2048 cores share a 40MB last-level cache [31], hence we can clearly see the pressure on the LLC when multiple tasks will compete for gaining access to it. This will lead to performance degradation when multiple tasks (or applications) will compete for these shared resources, and these potential contentions must be taken into account to obtain good co-scheduling performance.

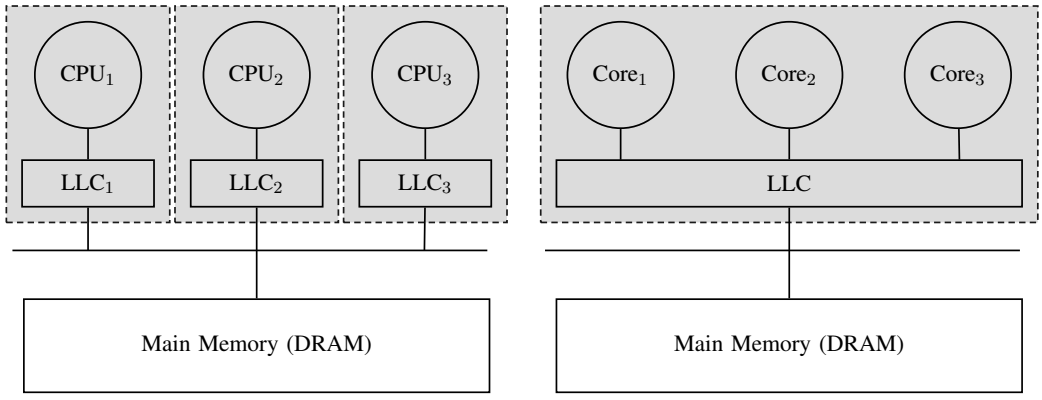
One of the questions at the core of this thesis is the following: from a given set of tasks that need to be executed, either independent tasks or tasks with dependencies, how to efficiently concurrently schedule these tasks on modern parallel architectures with different memory systems? In this thesis, we study scheduling algorithms on hardware-managed and software-managed scratchpad memory systems.

1.1.1 Parallel architectures

One of the first parallel architectures that appeared in the early 1960s was the symmetric multiprocessing system (SMP) [120]. SMP architectures were composed of multiple identical processors, each of them with its own cache, connected through a shared bus to the main memory (DRAM). In SMP architectures, all processors are considered independent, a performance of task running on such a processor will not be affected by other tasks running on neighbor processors.

Many studies have been conducted to schedule tasks onto SMPs [64, 66]. Scheduling on SMP systems consists in time-sharing the execution of tasks onto available processors, in other words, to decide when to schedule a task. But SMP architectures are poorly scalable because of the bottleneck arising from the bus between the processors and the main memory.

In 1996, Olukotun et al. [91] proposed a new microprocessor design, the single-chip multiprocessor (CMP). The idea is to embed all computing cores and cache memories on a single chip to reduce communication delays between cores, and to improve parallel performance. Since CMP systems consist of multiple cores on the same package, each core shares some elements as the last-level cache or the memory channels with the other cores. The key difference between SMP and CMP architectures lies in the fact that CMP cores are not independent (see Figure 1.1). On an SMP platform, the performance of a task running on a given processor is not affected by the tasks running on other processors. This is not the case for chip multiprocessors, as critical resources are shared by every core. Alternative designs can also be envisaged, in 2007, Vangal et al. [117] proposed a tiled approach called Intel Polaris with 80 tiles connected through a network-on-chip (NoC), where each tile has its own compute core and private cache.



(a) Symmetric multiprocessing (SMP) (b) Chip multiprocessor (CMP)

Figure 1.1: Comparison between SMP and CMP architectures.

Therefore, the advent of chip multiprocessors brings a new dimension in scheduling decisions: *space*. CMP schedulers must not only decide *when* to schedule a task but also *where*, on which core, to schedule a task. When CMPs started to be popular, many researchers used schedulers designed for SMP systems considering that CMPs can be seen as SMPs (each core is considered to be independent). This optimistic assumption turned out to be false, and led to severe degradations in scheduling performance. Resource contention can happen in cache memories, memory controllers and prefetching units [28, 74, 98]. Many studies showed that running two tasks on neighboring cores can lead to a severe global performance degradation, compared to running each task alone, by an important factor (up to three for worst cases!) due to resource contentions [16, 53, 109, 125]. Among resource contentions, cache contention is prominent [125].

Current many-core architectures [31, 61] can be considered as CMP platforms. Because these platforms exhibit a higher number of cores than classical CMPs, the resource contention effect is amplified on these platforms. To illustrate that the contention problem will not get better in the future, the best example is the PEZY-SC2 platform: where 2048 cores (possibly $16,384$ threads!) share a 40MB last-level cache [31]. In the worst case with $16,384$ threads, each thread has roughly access to 2KB of the last-level cache. Besides contentions due to cache sharing, another possible source of contentions is the new *high bandwidth memories* (HBM) embedded on some many-cores, as the Xeon Phi KNL [61]. These new HBM memories are scratchpad memories shared by all cores. For a given task running on such platform and using the high-speed bandwidth, its I/0 performance will be highly correlated to the communication pattern of other tasks running at the same time on the platform. We have addressed this problem in the last chapter of the thesis.

In addition of these several contention issues, supercomputers composed of multiple CMPs, each of them with hundreds of cores, are facing another challenging problem: *resilience*. Indeed, this increasing number of cores implies critically severe fault-tolerance issues. This is because the mean time between failures (MTBF) decreases linearly with the number of processors [57]. In this thesis, we address the resilience challenge in a co-scheduling context. Taking into account resource contentions and resiliency issues, while tasks are concurrently scheduled, is a key to maximize the efficiency and the usability of such parallel platforms.

1.1.2 Scratchpad memory systems

The well-known cache memories belong to a larger category of memories: the *scratchpad memories*. In the last chapter of this thesis, we investigate the problem of task graph scheduling on a particular CMP platform with a software-managed scratchpad memory.

Scratchpad memories (SPM) are a category of memories that are embedded on chips, near the processing units, in contrast to off-chip memories as DRAM that need a bus to communicate with the processing units. This kind of memory presents the advantage of being extremely fast to access with higher bandwidth, at the price of smaller size. A SPM can either be software or hardware controlled, the best known hardware-managed SPM are the cache memories. Cache memories are controlled by the hardware, a programmer cannot decide to allocate a specific data in cache. In opposition to hardware-managed memories, software-managed SPMs, as in modern GPUs or in recent many-core architectures, have to be managed manually by programmers. For example, the Intel Knights Landing (KNL) many-core processor [61] proposes an high-bandwidth SPM of 16 GB. A software-managed SPM can be seen as an addressable space by the programmer, the programmer can allocate any data in that memory at any time.

Software-managed scratchpad memories offer several advantages over hardware-managed memories. Directly-managed SPMs remove the notion of memory interferences, hence there is no need of using partitioning techniques to reduce interferences because a task cannot evict data of another task in that memory, it is the responsibility of the programmers or the OS to manage the available memory. Predicting the execution time of a task sharing a cache is challenging, but without complex replacement policies software-managed SPMs make the execution time of a task much more predictable. In addition, software-managed SPMs are also more energy efficient than hardware-managed SPMs like caches, because they do not use complicated hardware units. In this thesis, we explore co-scheduling and scheduling problems for hardware-managed SPM (with a focus on the last-level cache) and software-managed SPMs (focused on the KNL architecture).

To decide which data should be allocated into the scratchpad memory and which data should stay in the main memory, the memory management unit (MMU) must be able to characterize which part of the

code to prioritize. Scratchpad memory allocation strategies can be divided in two approaches: (i) static approaches where data are allocated into the SPM once at the beginning and the allocation do not change at runtime [4, 10], and (ii) dynamic approaches in which data allocated into the SPM may change during the execution of the task [30, 41]. Many static approaches use compiler techniques to determinate which are the most used code parts, hence data allocation is decided at compile time. Angiolini et al. [5] use a static strategy, at compile time, to detect heavily used arrays in the code and allocate them into the SPM to minimize off-ship communications. On the dynamic side, Egger et al. [42] profile tasks using the page fault mechanism inside the MMU, and from these information decide which data should be in the SPM, under the objective of minimizing energy consumption.

However, all these approaches are fine-grain, often focusing on code analysis, while our focus is more on coarse-grained approaches, basically at the task level. Another problem is that SPM are massively used in the embedded world, but not so much in the HPC world. One of the first architecture used in HPC with a software-managed scratchpad memory is the Intel KNL [61]. Perarnau et al. [97] showed that the performance of a simple stencil benchmark on that kind of architecture can be improved by using a scheme similar to out-of-core algorithms.

1.1.3 Concurrent scheduling

In the early 1980s, many operating systems (OS) scheduling policies were making an important assumption: processes running on the platform were independent, meaning that interactions as communications between processes, were an exception. With the advent of parallel architectures and parallel programming paradigms, this assumption turned out not to hold any longer, and it led scheduling policies to produce schedules with breaks and waiting time when processes were not independent. In 1982, to face these changes, Ousterhout [93] has introduced a new notion, called *co-scheduling*. A task is co-scheduled if all the processes (or threads) of this task, are executed simultaneously on different processors. The idea behind co-scheduling is to execute all processes of a task at the same time to minimize the waiting time due to inter-process communications. This novel idea has been implemented by Ousterhout et al. [94] in Medusa, an experimental operating system.

In his work [93], Ousterhout assumed independent processors. Given the period of this work, in 1982, this assumption made sense, but as discussed previously, in Section 1.1.1, current cores in CMP architectures are not independent and share some crucial parts, like caches and memory controllers. Thus, the scheduler must know on which cores to co-schedule each task to avoid contentions on shared resources. A possible solution to avoid interferences is to not co-schedule tasks, in other words, only schedule one task at a time on a CMP. As mentioned in the introduction, this solution is unrealistic due to the inherent sequential fraction of a task that limits the scalability and thus the efficiency of this solution. Many studies [28, 78, 83, 85, 109, 125] showed experimentally that the execution time of a thread can vary greatly depending on which other threads are running and sharing the resources at the same time. There are multiple sources of contentions, caches, memory controllers or prefetching units, and modeling all these interactions is a challenging endeavor.

Before reviewing the literature on interference models in the next section, we first discuss several co-scheduling strategies where some researchers assumed that all degradations factors are known beforehand. Jiang et al. [63] have proposed an interesting solution based on building a complete graph of *co-run degradations* between each task. A co-run degradation between two tasks is the ratio when both tasks share the cache compared to running solo. This study proposes an optimal co-schedule algorithm that minimizes the degradations assuming that all co-run degradations are known beforehand. A co-schedule consists in finding a mapping from threads to cores on a machine with multiple clusters sharing a last-level cache such that the resulting degradations are minimized. Although this solution

is interesting, it has severe limitations. Building such a complete graph of degradations is not possible when the number of cores and the number of tasks increases. Furthermore, each task is considered sequential and each task has the same length, adding the assumption of parallel tasks make the problem much more complicated.

Tian et al. [113] extended the previous theoretical study of Jiang et al. [63] by taking into account tasks of different lengths and by adding the possibility of rescheduling a task when some cores become available. In their work, they formulate the problem of finding an optimal co-schedule as a tree-search problem and used A*-based approaches to prune the co-scheduling search space, which is exponential in the number of tasks, and to approximate the optimal solution.

Snavely et al. [107, 108] have developed a *symbiotic* approach that does not need any knowledge on the tasks that are going to be scheduled. The term *symbiotic* comes from biology and indicates the mutual potential benefit that two biological organisms can obtain by living closely together and sharing some resources. The idea behind symbiotic co-scheduling is to randomly perturb the set of co-scheduled tasks and, by sampling hardware performance counters, to determinate which set of tasks maximizes the throughput of the platform. Besides the intrinsic difficulty of scheduling problems, in order to make co-scheduling approaches efficient, a major challenge is to find a way to model all these interactions on shared resources.

1.1.4 Cache contention models

Several sources of resource contention, caches, memory controllers or hardware prefetching units, are co-existing in a chip multiprocessors platform. According to several studies [51, 65, 75, 98, 111] contention in the last-level cache (LLC) appears to be one of the most critical sources of contention. We call cache contention, the fact that a task suffers from extra cache misses from other tasks running on the same platform. Indeed, other tasks bring their own data in the shared cache and may evict data from the original task. One first problem is that the LRU cache replacement policy, often used to manage cache memories, is not designed for concurrent accesses [65, 109]. The LRU policy is designed to take advantage of temporal locality by keeping in cache the most recently accessed cache lines. However, LRU is not designed to manage multiple concurrent accesses, LRU handles all concurrent accesses (for example from *threads*) uniformly, i.e., the cache policy is not aware of which thread would benefit the most from extra cache space [62]. To mitigate resource contention, the scheduler must be able to predict performance when multiple tasks are running at the same time and share the resources.

Many researchers have focused their efforts on finding techniques to predict the performance of multiple tasks sharing cache and they categorize task behaviors. The best known approaches are the stack distance profiles (SDP) [28, 50] and the miss rate curves (MRC) [110, 111]. The SDP is basically the distribution of cache hits among the LRU stack, it captures the temporal reuse behavior of cache for one thread at a time. An important assumption is made here: a SDP profile is assumed to be the same with or without sharing the cache with other threads. We consider $N + 1$ counters, $C_1, \ldots, C_N, C_{>N}$ for a N-ways associative LRU cache where the $C_{>N}$ counter is for cache misses. Then, for each cache access, one counter is incremented as follows: C_i is incremented if there is a cache hit at the i^{th} line in the LRU stack (hence i represents the distance in the stack). The first line in the LRU stack represents the most recently used cache line, the last is the least recently used. From the obtained histogram, scheduler policies are able to determinate if a thread has a good temporal behavior, i.e., if it cache accesses often touch first lines in the LRU stack. SDPs can be obtained statically at compile time or by running a task alone on the platform and recording each cache access for a fixed period of time. From a SDP we can derive a miss rate curve that represents the cache miss rate for a thread as a function of the cache size [109]. MRCs are used to characterize the behavior of tasks and provide information

on how a task will benefit from extra cache memory [111]. MRCs can be efficiently computed using hardware performance counters as showed by Tam et al. [110], although this method strongly depends on the architecture. SDPs and MRCs are designed to be used in a single-thread context, predicting the performance of multiple threads sharing a cache is much more challenging. Chandra et al. [28] designed an algorithm to merge two SDPs into a single profile that quantifies the extra cache misses when two threads share the LLC. Besides stack distance profiles and miss rate curves, another possibility to estimate the potential benefit from additional cache is to use an analytical model. Hartstein et al. [54] showed, with the Power Law of cache misses (or the $\sqrt{2}$ rule), how the cache size affects the cache miss ratio. The Power Law states that, if for a baseline cache of size C_0, the cache miss ratio is equal to m_0, then for a cache of size C, the cache miss ratio $m = m_0 \left(\frac{C_0}{C}\right)^{\alpha}$, where α is usually set to 0.5. Rogers et al. [101] used the Power Law of cache misses to model memory traffic to analyze the effect of bandwidth scaling in CMPs.

When the scheduler is able to approximate the potential performance degradation when multiple tasks or threads share the cache, this information can be used to build contention-aware schedules. Several approaches are possible, the best known example is an isolation approach called *cache partitioning*, where researchers partition the last-level cache into partitions such that the interferences between competing threads are minimized. Indeed, a task can only allocate cache lines in its partition. Among existing works, two major trends can be identified: studies arguing for *cache partitioning* or for *task classification*.

Multiple cache partitioning schemes have been designed, through both hardware techniques [16, 65, 86, 98] and software techniques [51, 75, 111, 112]. Most of the hardware approaches are efficient with a very low overhead at the execution time, but they suffer from an extra cost in terms of hardware components. Furthermore, these hardware solutions are difficult to implement and often only tested through simulated architectures. Qureshi et al. [98] propose hardware solution called the utility-based cache partitioning (UCP). UCP monitors, at runtime, the benefit of using cache (called the utility) for each task, through a low-overhead hardware circuit. Based on this information, UCP divides the cache among tasks to give more cache to high priority tasks. On the side of software-based solutions, the most popular is the *page coloring* solution, where physical pages are selected for task allocations so that they end up in specific sections of the cache [106]. Tam et al. [111], showed that important gains can be achieved through a static partitioning of the L2 cache using page coloring. Besides static strategies, dynamic cache partitioning strategies using page coloring have also been studied. In [75], the cache partitioning is refined and adjusted periodically at runtime, with the objective to maximize platform efficiency. Some solutions also concentrate their efforts on latency-sensitive tasks. Mars et al. [78] designed a runtime to improve QoS and fairness in batch scheduling for latency-sensitive tasks. This runtime handles what they call cross-core interference with a custom solution that maximize the utilization the platform, where utilization is the averaged ratio of effective running time of each task. Other resources can be partitioned using the same idea, like the translation look-aside buffer (TLB) [112] or the memory channels [83]. Some researchers have proposed solutions to address the challenge of building scheduling policies that take into account multiple resource contentions. Bitirgen et al. [15] use a machine learning approach, through artificial neural networks (ANN), to estimate the performance of each task running on the CMP at runtime. They propose a coordinated approach taking into account multiple contention sources (cache, bandwidth and power management) and, by comparing with uncoordinated scheduling policies (one contention considered) from the literature [85, 98], show that a coordinated approaches perform better than uncoordinated one. However, the use of multiple hardware ANNs per each task limits drastically the scalability of such approach.

Although cache partitioning solutions are known since many years, most of the previously cited studies on cache partitioning, used custom simulators or hardware prototypes. The first commercial and

widely available technique to partition the cache is the *Cache Allocation Technology* (CAT) [87] released by Intel in 2015. CAT is a technology that can partition several shared resources among cores, such as the LLC or the bandwidth, on the supported CPUs. This technology is the first to effectively partition the cache without any software or hardware modifications. Recently, Lo et al. [76] used this novel technique to isolate latency-sensitive tasks and thus obtain a safe collocation of tasks according a given Quality of Service.

Besides cache isolation, another solution is to categorize tasks based on their memory behaviors [67, 121]. The idea is, instead of partitioning the resources, to co-locate only tasks that have compatible behaviors. McGregor et al. [80] manage multiple threads by building pairs of threads that run together on a CMP based on performance information collected at runtime. They co-locate a thread that is memory-intensive with a thread that is more compute-intensive to balanced the pressure on memory controllers and on cache accesses.

1.2 Problematics and contributions

We have reviewed the differences between several parallel architectures, some co-scheduling approaches in the literature, and the different techniques used to model resource contention, in particular the contention in the last-level cache, which is at the core of this thesis. In this thesis, we focus on co-scheduling algorithms on CMP platforms in a high performance computing context. High performance computing tasks obey to some particularities, as their sequential fractions or their memory behaviors, that are interesting to take into account for co-scheduling algorithms. From classical CMPs that currently run on supercomputers, to emerging many-core architectures with new memories that will power future exascale supercomputing platforms, we take into account different memory constraints with the same objectives of maximizing the performance of co-scheduling or classical scheduling algorithms for HPC tasks.

1.2.1 Co-scheduling with cache partitioning

As emphasized below, several resource contentions on CMP platforms can dramatically degrade scheduling performance. In this thesis, we exclusively focus on last-level cache (LLC) interferences, indeed interferences in the cache are one of the most important degradations possible [125]. As we study models, we focus on static cache partitioning and scheduling solutions. Contrarily to dynamic schedulers, that do not know the behavior of the scheduled tasks, in this thesis we target static schedulers. Static schedulers rely on task knowledge such as estimated workload, speed-up profile or cache miss rate. We use the classic idea of cache partitioning to reduce the induced interferences, the novelty of our work lies in proposing a tractable theoretical model to analyze the co-scheduling performance of several (understand more than two) HPC applications sharing the last-level cache. The few theoretical models in the literature suffer from severe limitations, mostly due to the combinatorial nature of the problem [63]. In addition, these models often assume that interferences are known beforehand and do not propose to quantify them. In this thesis, we propose a complete analytical model to study static co-scheduling performance of applications on a shared LLC. The last originality of our work is to focus on HPC tasks that obey to Amdhal's law. Indeed very few studies on cache partitioning have adopted an HPC focus, most of them evaluated their solutions with classical benchmarks like SPEC, that do not exhibit HPC behaviors. Perarnau et al. [96] propose a tool to partition the cache, using page coloring, and show that cache partitioning can greatly improve performance on a multi-grid stencil computations, which is a widely used HPC kernel.

After a theoretical contribution with a complete co-scheduling performance model, taking into account cache contentions, the second contribution of this thesis is an experimental study. Many studies on scheduling with resource contentions evaluated their hardware or software solutions, through simulators [15, 28, 65, 83, 86, 98, 112, 121]. This is mainly due to the fact that, before 2015, they were no existing solution to easily partition the cache between concurrent tasks. A large amount of studies target specifically scheduling problems in operating systems, hence they often modify the kernel scheduler or the memory subsystem to verify their ideas [67, 75, 111]. However this thesis is focused on HPC tasks, then helped by the theoretical insights gained from our first contribution, we conduct an important experimental campaign on co-scheduling algorithms on a cache partitioned platform. We take advantage of the recent apparition of the CAT cache allocation technology [87] released by Intel, to partition the cache without modifying the underlying hardware nor the OS. With the help of the CAT, we provide the first, to the best of our knowledge, experimental study that clearly evaluates the impact of cache partitioning when co-scheduling multiple widely used HPC benchmarks.

1.2.2 Co-scheduling with resilience

After studying the impact of interferences in the last-level cache for co-scheduling algorithms, we are interested in another aspect of co-scheduling on massively parallel architectures. Indeed, the two first contributions of this thesis deal with the impact of the high number of cores in current parallel architectures from an interference point of view. The co-scheduling approach has been proved to be efficient, but only in a fault-free context. In this thesis, we extend a previous work [9] by taking into account the resiliency parameter. To the best of our knowledge, this third contribution is the first study on a co-scheduling problem in a resilient context. Note that, in this work, we do not take into account any interference phenomenon to keep the problem tractable.

1.2.3 Scheduling for emerging parallel architectures

Our last contribution aims at building a first-step study about scheduling a task graph on the emerging many-core architectures that exhibit a new on-package high bandwidth memory (HBM). The problem of optimizing scheduling algorithms for systems with two different memories (also called *out-of-core*) is old [103]. We call a platform with a high-bandwidth memory on-chip, a *deep memory* platform. Deep memory platforms are not exactly following out-of-core models, in the out-of-core model a data must be read from the slow memory before being allocated into the fast memory, but with HBMs the processor can read either from fast or slow memory. Chandrasekar et al. [29] discussed a runtime method to schedule tasks with data dependencies on a dual-memory platform (i.e., a main memory and a scratchpad memory). Unfortunately, the scheduling algorithm is limited to scheduling a task only after all its input data has been moved to faster memory. Also, no theoretical analysis of this scheduling heuristic was performed.

Our last contribution extends a previous work [97] by proposing a complete performance model for task graphs scheduling on deep-memory architecture. To the best of our knowledge, no comprehensive study has addressed memory movement and task scheduling for these new deep-memory architectures from a performance-model standpoint.

Chapter 2

Co-scheduling applications on cache-partitioned systems

As emphasized in the introduction, at scale, the massive concurrency and the I/O movements of high performance computing (HPC) applications are expected to be one of the most critical problems [99]. Observations on the Intrepid machine at Argonne National Laboratory (ANL) show that I/O transfers can be slowed down up to 70% due to congestion [47]. When ANL upgraded its house supercomputer from Intrepid (Peak perf: 0.56 PFlops; peak I/O throughput: 88 GB/s) to Mira (Peak perf: 10 PFlops; peak I/O throughput: 240 GB/s), the net result for an application whose I/O throughput scales linearly (or worse) with performance was a downgrade from 160 GB/PFlop to 24 GB/PFlop!

To cope with such an imbalance (which is not expected to reduce on future platforms), a possible approach is to develop *in situ* co-scheduling analysis and data preprocessing on dedicated nodes [99]. This scheme applies to data-intensive periodic workflows where data is generated by the main simulation, and parallel processes are run to process this data with the constraints that output results should be sent to disk storage before newly generated data arrives for processing. These solutions are starting to be implemented for HPC applications. Sewell et al. [104] explain that in the case of the HACC application (a cosmological code), petabytes of data are created to be analyzed later. The analysis is done by multiple independent processes. The idea of their work is to minimize the amount of data copied to I/O filesystem, by performing the analysis at the same time as HACC is running (what they call *in situ*). The main constraint is that these processes are data-intensive and are handled by a dedicated machine. Also, the execution of these processes should be done efficiently enough so that they finish before the next batch of data arrives, hence resulting in a pipelined approach. All these frameworks motivate the design of efficient co-scheduling strategies.

As detailed in the introduction, one main issue of co-scheduling is to evaluate *co-run degradations* due to cache sharing [125]. Many studies have shown that interferences on the shared last-level cache (LLC) can be detrimental to co-scheduled applications [73]. Previous solutions consisted in preventing co-schedule of possibly interfering workloads, or terminating low importance applications [123]. Lo et al. [76] recently showed experimentally that important gains could be reached by co-scheduling applications with strict cache partitioning enabled. Cache partitioning, the technique at the core of this work, consists in reserving exclusivity of subsections of the LLC of a chip multi-processor (CMP), to some of the applications running on this CMP. This functionality was recently introduced by Intel under the name *Cache Allocation Technology* [60]. With the advent of large shared memory multi-core machines (e.g., Sunway TaihuLight, the current #1 supercomputer uses 256-cores processor chips with a shared memory of 32GB [35]), the design of algorithms that co-schedule applications efficiently and decide how to partition the shared memory (seen as the cache here), is becoming critical.

In this chapter, we study the following problem. We are given a set of Amdahl applications, i.e., parallel applications obeying Amdahl's speedup law [3] (see Equation 2.1 for details). Amdahl's law has had a profound impact on the evolution of HPC [56] and many scientific applications, including most NAS Parallel Benchmarks, obey this law [25]. We are also given a multi-core processor with a shared last-level cache LLC. How can we best partition the LLC to minimize the total execution time (or *makespan*), i.e., the moment when the last application finishes its computation. For each application, we assume that we know the number of compute operations to perform, and the miss rate on a fixed size cache. For the multi-core processor, we know its LLC size, the cost for a cache miss, the cost for a cache hit, the size of the cache and total number of processors. For the theoretical study, we assume that these processors can be shared by two applications through multi-threading [68], hence we can assign a rational number of processors to each application, and this allows us to study the intrinsic complexity of co-scheduling with cache partitioning. Equipped with all these applications and platform parameters, recent work [54, 68, 101] shows how to model the impact of cache misses and to accurately predict the execution time of an application.

Main contributions. In this chapter, we show that, with rational numbers of processors, the co-scheduling problem is NP-complete, even when applications are perfectly parallel, i.e., their speed-up scales up linearly with the number of processors. We show several results that characterize optimal solutions, and in particular that the co-scheduling cache-partitioning problem reduces to deciding which subset of applications will share the LLC; when this subset is known, we show how to determine the optimal cache fractions and rational number of processors for perfectly-parallel applications. Furthermore, we show that all applications should finish at the same time, even if they are not perfectly parallel. These theoretical results guide the design of heuristics for Amdahl applications. We show through extensive simulations (using both rational and integer numbers of processors) that our heuristics greatly improve the performance of cache-partitioning algorithms, even for parallel applications obeying Amdahl's law with a large sequential fraction, hence with a limited speedup profile.

The rest of this chapter is organized as follows. Section 2.1 provides an overview of related work. Section 2.2 is devoted to formally defining the framework and all model parameters. Section 2.3 gives our main theoretical contributions. The heuristics are defined in Section 2.4, and evaluated through simulations in Section 2.5. Finally, Section 2.6 outlines our main findings and discusses directions for future work.

2.1 Related work

In this section, we review the related work on co-scheduling and cache partitioning studies. Note that this survey is also relevant to Chapter 3.

2.1.1 Co-scheduling and interferences

Since the advent of systems with tens of cores, co-scheduling has received considerable attention. We refer to [34, 76, 83] for a survey of many approaches to co-scheduling. The main idea is to execute several applications concurrently rather than in sequence, with the objective to increase platform throughput. Indeed, some individual applications may well not need all available cores, or some others could use all resources, but at the price of a dramatic performance loss. In particular, the latter case is encountered whenever application speedup becomes too low beyond a given processor count. A new trend in large-scale simulations are *in-situ* and *in-transit* approaches, to visualize and analyze the data during the

simulation [38]. Basically, the idea behind these approaches is that a new dataset is generated periodically, and we need to run different applications on different parts of this dataset before the next period. In the *in-situ* approach, simulation and analyzes are co-located in the same node, while in the *in-transit* approach, the data analyzes are outsourced onto dedicated nodes [13]. Several studies have shown that large-scale simulations with *in-situ* could benefit from co-scheduling approaches [12, 104]. The difficulty consists in ensuring that the in-situ part processes the data fast enough to avoid slowing down the main simulation, which is directly related to co-scheduling issues: how to partition the resources across the concurrent analysis applications that share the CMP?

Indeed, when executing simultaneously, any two applications will compete for shared resources, which will create interferences and decrease their throughput. Modeling application interference is a challenging task. Dynamic schedulers are used when application behavior is unknown [98, 113]. Static schedulers aim at optimizing the sharing of the resources by relying on application knowledge such as estimated workload, speed-up profile, cache behavior, etc. One widely-used approach is to build an interference graph whose vertices are applications and whose edges represent degradation factors [55, 63, 124]. This approach is interesting but hard to implement. Indeed, the interaction of two applications depends on many factors, such as their size, their core count, the memory bandwidth, etc. Obtaining the speedup profile of a single application already is difficult and requires intensive benchmarking campaigns. Obtaining the degradation profile of two applications is even more difficult and can be achieved only for regular applications. To further darken the picture, the interference graph subsumes only pairwise interactions, while a global picture of the processor and cache requirements for all applications is needed by the scheduler.

Shared resources include cache, memory, I/O channels and network links, but among potential degradation factors, cache accesses are prominent [126]. When several applications share the cache, they are granted a fraction of cache lines as opposed to the whole cache, and their cache miss ratio increases accordingly. Hartstein et al. [54] showed, with the Power Law of cache misses (or the $\sqrt{2}$ rule), how the cache size affects the cache miss ratio. The Power Law states that, if for a baseline cache of size C_0, the cache miss ratio is equal to m_0, then for a cache of size C, the cache miss ratio $m = m_0 \left(\frac{C_0}{C}\right)^{\alpha}$, where α is usually set to 0.5. To reduce these interferences we use, in this thesis, a technique called *cache partitioning*. Cache partitioning, consists in reserving exclusivity of subsections of the last-level cache of a chip multi-processor (CMP), to some of the applications running on this CMP.

2.1.2 Cache partitioning techniques

Multiple cache partitioning schemes have been designed, through hardware techniques [16, 65, 86, 98] and software techniques [51, 75, 111, 112]. Most of the hardware approaches are efficient with a very low overhead at the execution time, but they suffer from an extra cost in terms of hardware components. In addition, hardware solutions are difficult to implement and often only tested through simulated architectures. An interesting hardware solution is the utility-based cache partitioning (UCP) [98]. UCP proposes to monitor, at runtime, the benefit of using cache (called utility) for each application, through a low hardware overhead circuit. Based on these information UCP divides the cache among applications to give more cache to high priority applications.

On the side of software-based solutions, the most popular is *page coloring*, where physical pages are selected for application allocations so that they end up in specific sections of the cache. Tam et al. [111], showed that important gains can be achieved through a static partitioning of the L2 cache using page coloring. Besides static strategies, dynamic cache partitioning strategies using page coloring have also been studied. In [75], the cache partitioning is refined and adjusted periodically at runtime,

with the objective to maximize platform efficiency. But recently, Intel released a new software technique to internally partition the last level cache (LLC), called the *Cache Allocation Technology* (CAT) [76, 87].

In this chapter, we focus on a static allocation of LLC cache fractions, and processor numbers, to concurrent applications as a function of several parameters (cache-miss ratio, access frequency, operation count). To the best of our knowledge, this work is the first analytical model and complexity study for this challenging problem.

2.2 Model

This section details platform and application parameters, and formally states the optimization problem.

2.2.1 Architecture

We consider a parallel platform of p homogeneous computing elements, or *processors*, that share two storage locations:

- A small storage $\mathcal{S}_s$ with low latency, governed by a LRU replacement policy, also called *cache*;
- A large storage $\mathcal{S}_l$ with high latency, also called *memory*.

More specifically, C_s (resp. C_l) denotes the size of $\mathcal{S}_s$ (resp. $\mathcal{S}_l$), and l_s (resp. l_l) the latency of $\mathcal{S}_s$ (resp. $\mathcal{S}_l$). In this work, we assume that $C_l = +\infty$. We have the relation $l_s \ll l_l$. In this work, we consider the cache partitioning technique [60], where one can allocate a portion of the cache to applications so that they can execute without interference from other applications.

2.2.2 Applications

There are n independent parallel applications to be scheduled on the parallel platform, whose speedup profiles obey Amdahl's law [3]. For an application T_i, we define several parameters:

- w_i, the number of computing operations needed for T_i;
- s_i, the sequential fraction of T_i;
- f_i, the frequency of data accesses of T_i: f_i is the number of data accesses per computing operation;
- a_i, the memory footprint of T_i.

We use these parameters to model the execution of each application as follows.

Parallel execution time

Let $Fl_i(p_i)$ be the number of operations performed by each processor for application T_i, when executed on p_i processors. According to Amdahl's speedup profile [3], we have

$$Fl_i(p_i) = s_i w_i + (1 - s_i)\frac{w_i}{p_i} \tag{2.1}$$

The power law of cache misses

In chip multi-processors, many authors have observed that the Power Law accurately models how the cache size affects the miss rate [54, 68, 101]. Mathematically, the power law states that if m_0 is the miss rate of a workload for a baseline cache size C_0, the miss rate m for a new cache size C can be expressed as $m = m_0 \left(\frac{C_0}{C}\right)^{\alpha}$ where α is the sensitivity factor from the Power Law of Cache Misses [54, 68, 101] and typically ranges between 0.3 and 0.7 with an average at 0.5. Note that, by definition, a rate cannot be higher than 1, hence we extend this definition as:

$$m = \min\left(1, m_0 \left(\frac{C_0}{C}\right)^{\alpha}\right). \tag{2.2}$$

This formula can be read as follows: if the cache size allocated is too small, then the execution goes as if no cache was allocated, and all accesses will be misses.

Computations and data movement

We use the cost model introduced by Krishna et al. [68] to evaluate the execution cost of an application as a function of the cache fraction that it has been allocated. Specifically, for each application, we define m_0, the miss rate of application T_i with a cache of size C_0 (we can also use the miss rate of applications with a cache of another fixed size). We express the execution time of T_i as a function of p_i, the number of processors allocated to T_i, and x_i, the fraction of $\mathcal{S}_s$ allocated to T_i (recall both are rational numbers). Let $Fl_i(p_i)$ be the number of operations performed by each processor for application T_i, given that the application is executed on p_i processors. We have $Fl_i(p_i) = s_i w_i + (1 - s_i)\frac{w_i}{p_i}$ according to Amdahl's speedup profile. Finally,

$$\mathcal{E}\text{xe}_i(p_i, x_i) = \begin{cases} Fl_i(p_i)\left(1 + f_i\left(l_s + l_l\right)\right) & \text{if } x_i = 0; \\ Fl_i(p_i)\left(1 + f_i\left(l_s + l_l \cdot \min\left(1, \frac{m_0}{\left(\frac{x_i C_s}{C_0}\right)^{\alpha}}\right)\right)\right) & \text{if } x_i C_s \leq a_i; \\ Fl_i(p_i)\left(1 + f_i\left(l_s + l_l \cdot \min\left(1, \frac{m_0}{\left(\frac{a_i}{C_0}\right)^{\alpha}}\right)\right)\right) & \text{otherwise.} \end{cases} \tag{2.3}$$

Indeed, for each operation, we pay the cost of the computing operation, plus the cost of data accesses, and by definition we have f_i accesses per operation. At each access, we pay a latency l_s, and an additional latency l_l in case of cache miss (see Equation 2.2). The last case states that we cannot use a portion of cache greater than the memory footprint a_i of application T_i. This model is somewhat pessimistic: cache accesses to the same variable by two different processors are counted twice. We show in Section 2.5 that despite this conservative assumption (no sharing), co-scheduling can outperform classical approaches that sequentially deploy each application on the whole set of available resources.

Equation 2.3 calls for a few observations. For notational convenience, let $d_i = m_0 \left(\frac{C_0}{C_s}\right)^{\alpha}$:

- It is useless to give a fraction of cache larger than $\frac{a_i}{C_s}$ to application T_i;
- Because of the minimum $\min\left(1, \frac{d_i}{(x_i)^{\alpha}}\right)$, either $x_i > d_i^{\frac{1}{\alpha}}$, or $x_i = 0$: indeed, if we give application T_i a fraction of cache smaller than $d_i^{\frac{1}{\alpha}}$, the minimum is equal to 1, and this fraction is wasted.

Hence, we have for all i:

$$x_i = 0 \quad \text{or} \quad d_i^{\frac{1}{\alpha}} < x_i \leq \frac{a_i}{C_s}. \tag{2.4}$$

Of course, if $d_i^{\frac{1}{\alpha}} \geq \frac{a_i}{C_s}$ for some application T_i, then $x_i = 0$. We denote by $\mathcal{E}\text{xe}_i^{\text{seq}}(x_i) = \mathcal{E}\text{xe}_i(1, x_i)$ the sequential execution time of application T_i with a fraction of cache x_i.

2.2.3 Scheduling problem

Given n applications $T_1, \ldots, T_n$, we aim at partitioning the shared cache and assign processors so that the concurrent execution of these applications takes minimal time. In other words, we aim at minimizing the execution time of the longest application, when all applications start their execution at the same time. Formally:

Definition 2.1 (CoSchedCache)**.** *Given n applications $T_1, \ldots, T_n$ and a platform with p identical processors sharing a cache of size C_s, find a schedule $\{(p_1, x_1), \ldots, (p_n, x_n)\}$ with $\sum_{i=1}^n p_i \leq p$, and $\sum_{i=1}^n x_i \leq 1$, that minimizes*

$$\max_{1 \leq i \leq n} \mathcal{E}xe_i(p_i, x_i).$$

We pay particular attention in the following to *perfectly parallel* applications, i.e., applications T_i with $s_i = 0$. In this case, $\mathcal{E}\text{xe}_i(p_i, x_i) = \frac{\mathcal{E}\text{xe}_i(1,x_i)}{p_i} = \frac{\mathcal{E}\text{xe}_i^{\text{seq}}(x_i)}{p_i}$. The co-scheduling problem for such applications is denoted CoSchedCachePP.

2.3 Complexity results

In this section, we focus on the CoSchedCache problem with rational numbers of processors in order to study the intrinsic complexity of co-scheduling with cache partitioning. We first prove that in an optimal execution, all applications must complete at the same time when using rational numbers of processors (Section 2.3.1). We remind that CoSchedCache is NP-complete, even for perfectly parallel applications (Section 2.3.2), and we show several dominance results on the optimal solution (Section 2.3.3). While some of these dominance results only hold for perfectly parallel applications, they will guide the design of heuristics for general applications in Section 2.4.

2.3.1 All applications complete at the same time

Lemma 2.1. *To minimize the makespan when using rational numbers of processors, all applications must finish at the same time.*

Proof. Consider n applications $T_1, \ldots, T_n$ that obey Amdahl's law, and a solution $\mathcal{S} = \{(p_i, x_i)\}_{1 \leq i \leq n}$ to CoSchedCache. Let $D_{\mathcal{S}} = \max_i \mathcal{E}\text{xe}_i(p_i, x_i)$ be the makespan of this solution. For simplicity, we let

$$\begin{aligned} A_i &= 1 + f_i \left(l_s + l_l \cdot \min \left(1, \frac{m_{1\text{MB}\mathcal{S}_s}^i}{\left(\frac{x_i C_s}{10^6}\right)^{\alpha}} \right) \right), \\ b_i &= A_i w_i s_i, \\ c_i &= A_i w_i (1 - s_i) \end{aligned}$$

Hence, $\mathcal{E}\text{xe}_i(p_i, x_i) = b_i + \frac{c_i}{p_i}$. The set of applications whose execution time is exactly $D_{\mathcal{S}}$ is denoted by $I_{\mathcal{S}}$.

We show the result by contradiction. We consider an optimal solution $\mathcal{S}$ whose subset $I_{\mathcal{S}}$ has minimal size (i.e., for any other optimal solution $\mathcal{S}_o$, $|I_{\mathcal{S}}| \leq |I_{\mathcal{S}_o}|$). Then we show that if $|I_{\mathcal{S}}| \neq n$, we can construct a solution $\mathcal{S}'$ with either (i) a smaller makespan if $|I_{\mathcal{S}}| = 1$ (contradicting the optimality hypothesis), or (ii) one less application whose execution time is exactly $D_{\mathcal{S}}$ (contradicting the minimality hypothesis).

Assume $|I_{\mathcal{S}}| \neq n$, let $T_{i_0} \in I_{\mathcal{S}}$ and $T_{i_1} \notin I_{\mathcal{S}}$. We have $\mathcal{E}\text{xe}_{i_1}(p_{i_1}, x_{i_1}) < \mathcal{E}\text{xe}_{i_0}(p_{i_0}, x_{i_0}) = D_{\mathcal{S}}$, that is

$$b_{i_1} + \frac{c_{i_1}}{p_{i_1}} < b_{i_0} + \frac{c_{i_0}}{p_{i_0}}, \text{ and hence } \quad (b_{i_1} - b_{i_0})p_{i_0}p_{i_1} - c_{i_0}p_{i_1} + c_{i_1}p_{i_0} < 0. \tag{2.5}$$

We now prove that we can always find $0 < \varepsilon < p_{i_1}$ s.t. $\mathcal{E}\text{xe}_{i_0}(p_{i_0}, x_{i_0}) > \mathcal{E}\text{xe}_{i_0}(p_{i_0} + \varepsilon, x_{i_0}) > \mathcal{E}\text{xe}_{i_1}(p_{i_1} - \varepsilon, x_{i_1})$, i.e.,

$$D_{\mathcal{S}} = b_{i_0} + \frac{c_{i_0}}{p_{i_0}} > b_{i_0} + \frac{c_{i_0}}{p_{i_0}+\varepsilon} > b_{i_1} + \frac{c_{i_1}}{p_{i_1}-\varepsilon}\,.$$

The left part of inequality $b_{i_0} + \frac{c_{i_0}}{p_{i_0}} > b_{i_0} + \frac{c_{i_0}}{p_{i_0}+\varepsilon}$ is always true when $\varepsilon > 0$. For the right part of inequality above, we have:

$$-(b_{i_1} - b_{i_0})\varepsilon^2 + [(p_{i_1} - p_{i_0})(b_{i_1} - b_{i_0}) + c_{i_0} + c_{i_1}]\varepsilon + (b_{i_1} - b_{i_0})p_{i_0}p_{i_1} - c_{i_0}p_{i_1} + c_{i_1}p_{i_0} < 0. \tag{2.6}$$

From Equation 2.5, we know that $(b_{i_1} - b_{i_0})p_{i_0}p_{i_1} - c_{i_0}p_{i_1} + c_{i_1}p_{i_0} < 0$, so we can always find a $0 < \varepsilon < p_{i_1}$ that could make Equation 2.6 satisfied.

Then clearly, $\mathcal{S}' = \{(p'_i, x_i)\}_i$ where p'_i is (i) p_i if $i \notin \{i_0, i_1\}$, (ii) $p_{i_0} + \varepsilon$ if $i = i_0$, (iii) $p_{i_1} - \varepsilon$ if $i = i_1$, is a valid solution: we have the property $\sum_i p'_i = \sum_i p_i \leq p$, and $\sum_i x'_i = \sum_i x_i \leq 1$.

Hence,

- If $|I_{\mathcal{S}}| = 1$, then for all i, $\mathcal{E}\text{xe}_i(p'_i, x_i) < D_{\mathcal{S}}$, hence showing that $\mathcal{S}$ is not optimal;
- Else, $I_{\mathcal{S}'} = I_{\mathcal{S}} \setminus \{i_0\}$, and $D_{\mathcal{S}'} = D_{\mathcal{S}}$, hence showing that $\mathcal{S}$ is not minimal.

This shows that necessarily, $|I_{\mathcal{S}}| = n$. □

2.3.2 Intractability

We prove that the problem is NP-complete, even for perfectly parallel applications. Therefore, we formally state the decision problem associated to CoSchedCachePP:

Definition 2.2 (CoSchedCachePP-Dec). *Given n perfectly parallel applications $T_1, \ldots, T_n$ and a platform with p identical processors sharing a cache of size C_s, and given a bound K on the makespan, does there exist a schedule $\{(p_1, x_1), \ldots, (p_n, x_n)\}$, where p_i and x_i are nonnegative rational numbers with $\sum_{i=1}^n p_i \leq p$ and $\sum_{i=1}^n x_i \leq 1$, such that $\max_{1\leq i\leq n} \mathcal{E}xe_i(p_i, x_i) \leq K$?*

For perfectly parallel applications, we can transform CoSchedCachePP into an equivalent problem that does not depend on the number of processors but that relies simply on the cache partitioning strategy (Lemma 2.3 below). This result will guide processor assignment for general applications in Section 2.4. We start with a few lemmas. The following lemma shows the optimal rational processor assignment:

Lemma 2.2. *Given n perfectly parallel applications $T_1, \ldots, T_n$ and a partitioning of the cache $\{x_1, \ldots, x_n\}$, then the optimal number of processors for application T_i ($i \in \{1, \ldots, n\}$) is:*

$$p_i = p \frac{\mathcal{E}xe_i^{seq}(x_i)}{\sum_{j=1}^{n} \mathcal{E}xe_j^{seq}(x_j)}.$$

Proof. According to Lemma 2.1, all applications finish at the same time. Given $i_0 \in \{1, \ldots, n\}$, we have $\frac{\mathcal{E}\text{xe}_{i_0}^{\text{seq}}(x_{i_0})}{p_{i_0}} = \frac{\mathcal{E}\text{xe}_i^{\text{seq}}(x_i)}{p_i}$ for all $1 \leq i \leq n$. In addition, we have $\sum_{i=1}^{n} p_i = p$: the fact that this bound is tight in an optimal solution is due to the fact that we have perfectly parallel applications. We express p in terms of the others variables, and we do the summation: $p = \sum_{i=1}^{n} p_i = \frac{p_{i_0}}{\mathcal{E}\text{xe}_{i_0}^{\text{seq}}(x_{i_0})} \sum_{i=1}^{n} \mathcal{E}\text{xe}_i^{\text{seq}}(x_i)$. This directly leads to the result. □

Lemmas 2.1 and 2.2 lead to the following reformulation of CoSchedCachePP:

Lemma 2.3. CoSchedCachePP *can be rewritten as finding the optimal cache partitioning strategy $\mathcal{X} = \{x_1, \ldots, x_n\}$ that minimizes the completion time of an optimal solution:*

$$\frac{1}{p} \sum_{i=1}^{n} \mathcal{E}xe_i^{seq}(x_i). \tag{2.7}$$

Proof. Lemma 2.2 gives us that in an optimal solution the processor distribution is uniquely determined by the cache partitioning strategy. Furthermore, given a cache partitioning strategy, we know that all applications finish at the same time (Lemma 2.1) and that the completion time is equal to

$$\frac{\mathcal{E}\text{xe}_1^{\text{seq}}(x_1)}{p_1} = \frac{\sum_{i=1}^{n} \mathcal{E}\text{xe}_i^{\text{seq}}(x_i)}{p}.$$

□

Theorem 2.1. CoSchedCachePP-Dec *is NP-complete.*

Proof. Building upon these lemmas, we can prove Theorem 2.1 by using a reduction from Knapsack, which is NP-complete [48].

CoSchedCachePP-Dec is obviously in NP: given the x_i's, it is easy to verify all constraints in linear time. We prove the completeness by a reduction from Knapsack, which is NP-complete [48]. Consider an arbitrary instance $\mathcal{I}_1$ of Knapsack: given n objects, each with positive integer size u_i and positive integer value v_i for $1 \leq i \leq n$, and two positive integer bounds U and V, does there exist a subset $I \subset \{1, \ldots, n\}$ such that $\sum_{i \in I} u_i \leq U$ and $\sum_{i \in I} v_i \geq V$? Given $\mathcal{I}_1$, we construct the following instance $\mathcal{I}_2$ of CoSchedCachePP-Dec:

- We define two constants $\varepsilon = \frac{1}{N(N+1)}$ and $\eta = 1 - \frac{1}{N}$, where $N = \max(n, 2U + 1)$.
- We let $d_i = (\frac{u_i \eta}{U})^{\alpha}$, $e_i = \left(d_i^{\frac{1}{\alpha}} + \varepsilon\right)^{\alpha}$, $a_i = e_i^{\frac{1}{\alpha}} C_s$, and $w_i f_i l_l = \frac{v_i}{1 - \frac{d_i}{e_i}}$ for $1 \leq i \leq n$. Note that we only need the value of the product $w_i f_i$, and we can set one of them arbitrarily.
- The bound K is defined as:

$$pK = \sum_{i=1}^{n} w_i (1 + f_i l_s) + \sum_{i=1}^{n} w_i f_i l_l - V.$$

To simplify notations, let $z_i = w_i f_i l_l$. Letting $A = \sum_{i=1}^{n} w_i (1 + f_i l_s)$ and $Z = \sum_{i=1}^{n} z_i$, we get $pK = A + Z - V$. Also, we have $\sum_{i=1}^{n} w_i \left(1 + f_i \left[l_s + l_l \cdot \min\left(1, \frac{d_i}{x_i^{\alpha}}\right)\right]\right) = A + B$, where $B = \sum_{i=1}^{n} z_i \min(1, \frac{d_i}{x_i^{\alpha}})$. Recall from Lemma 2.3 that $\mathcal{I}_2$ has a solution if and only if $\frac{1}{p}(A + B) \leq K$.

Let $I_C \subseteq \{1, \ldots, n\}$ denote the subset of applications that are given some cache ($x_i \neq 0$ if and only if $i \in I_C$). We also call I_C the nonzero subset of $\mathcal{I}_2$. We have

$$d_i^{\frac{1}{\alpha}} \leq x_i \leq \frac{a_i}{C_s} = e_i^{\frac{1}{\alpha}},$$

so that we can rewrite $B = Z - \sum_{i \in I_C} z_i \left(1 - \frac{d_i}{x_i^\alpha}\right)$. Given the value of the bound K, we have $A + B \leq pK$ if and only if

$$\sum_{i \in I_C} z_i (1 - \frac{d_i}{x_i^\alpha}) \geq V.$$

We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution subset $I \subset \{1, \ldots, n\}$. Then we let $x_i = e_i^{\frac{1}{\alpha}}$ if $i \in I$ and $x_i = 0$ otherwise. This is a valid solution to $\mathcal{I}_2$ with nonzero subset $I_C = I$. Indeed:

- If $i \in I$, then $d_i^{\frac{1}{\alpha}} \leq x_i = e_i^{\frac{1}{\alpha}} = \frac{a_i}{C_s}$.
- We have
$$\sum_{i \in I} x_i = \sum_{i \in I} (d_i^{\frac{1}{\alpha}} + \varepsilon) = \sum_{i \in I} \frac{u_i \eta}{U} + |I|\varepsilon.$$
But $\sum_{i \in I} \frac{u_i \eta}{U} \leq \eta$ (since we have a solution for $\mathcal{I}_1$), and $|I|\varepsilon \leq n\varepsilon \leq \frac{1}{N+1}$, hence $\sum_{i \in I} x_i \leq \eta + \frac{1}{N+1} \leq 1$.
- Finally, $\sum_{i \in I} z_i (1 - \frac{d_i}{x_i^\alpha}) = \sum_{i \in I} z_i (1 - \frac{d_i}{e_i}) = \sum_{i \in I} v_i \geq V$ (since we have a solution for $\mathcal{I}_1$), hence $A + B \leq pK$.

Suppose now that $\mathcal{I}_2$ has a solution, and let I_C be its nonzero subset. We claim that $I = I_C$ is a solution to $\mathcal{I}_1$. Indeed, for $i \in I_C$ we have $d_i \leq x_i^\alpha \leq e_i$ and $\sum_{i \in I_C} z_i (1 - \frac{d_i}{x_i^\alpha}) \geq V$. First, we have $\sum_{i \in I_C} z_i (1 - \frac{d_i}{x_i^\alpha}) \geq \sum_{i \in I_C} z_i (1 - \frac{d_i}{e_i}) = \sum_{i \in I_C} v_i$, hence $\sum_{i \in I_C} v_i \geq V$. Then $\sum_{i \in I_C} d_i^{\frac{1}{\alpha}} \leq \sum_{i \in I_C} x_i \leq 1$, and $\sum_{i \in I_C} d_i^{\frac{1}{\alpha}} = \sum_{i \in I_C} \frac{u_i \eta}{U}$, hence $\sum_{i \in I_C} u_i \leq \frac{U}{\eta}$. But $\frac{U}{\eta} \leq U + \frac{1}{2}$ by the choice of η, thus $\sum_{i \in I_C} u_i \leq U + \frac{1}{2}$. Because the sizes are integers, $\sum_{i \in I_C} u_i \leq U$. Altogether, I_C is indeed a solution to $\mathcal{I}_1$. This concludes the proof. □

2.3.3 Dominance results for perfectly parallel applications

In this section, we provide dominance results that will guide the design of heuristics. The dominance results are for perfectly parallel applications ($s_i = 0$) but we give intuition on how to extend this work for Amdahl applications in Section 2.3.4. Finally, we further assume that application memory footprints are larger than the cache size ($a_i = +\infty$), and we assume rational numbers of processors.

The core of the previous intractability result relies on the hardness to determine the set of applications that receive a cache fraction (denoted by I_C) and those that do not (denoted by $\overline{I_C}$). In this section, we show (i) how to determine the optimal solution when these sets I_C and $\overline{I_C}$ are known, and (ii) whether one can disqualify some partitions as being sub-optimal.

In particular, we define a set of partitions of applications that we call dominant (Definition 2.4). We show that (i) if a partition of applications $I_C, \overline{I_C}$ is dominant, then we can compute the minimum execution time for this partition, and (ii) if a partition is not dominant, then we can find a better dominant partition. We start by rewriting the problem when the partitioning $I_C, \overline{I_C}$ of applications is known:

Definition 2.3 (CSCPP-PART$\left(I_C, \overline{I_C}\right)$)**.** *Given a set of applications $T_1, \ldots, T_n$ and a partition $I_C, \overline{I_C}$, the problem* CSCPP-PART$\left(I_C, \overline{I_C}\right)$ *(for* COSCHEDCACHEPP-PART*) is to find a set $\mathcal{X} = \{x_1, \ldots, x_n\}$ that minimizes the execution time:*

$$\frac{1}{p}\left(\sum_{i\in\overline{I_C}} w_i(1 + f_i(l_s + l_l)) + \sum_{i\in I_C} w_i(1 + f_i l_s + f_i l_l \frac{d_i}{x_i^\alpha})\right)$$

under the constraints $x_i = 0$ if $i \in \overline{I_C}$, $x_i > d_i^{1/\alpha}$ if $i \in I_C$, and $\sum_{1\leq i\leq n} x_i \leq 1$.

We now relax some bounds in CSCPP-PART$\left(I_C, \overline{I_C}\right)$ and define CSCPP-EXT$\left(I_C, \overline{I_C}\right)$, which is the same problem except that the constraints on the x_i's when $i \in I_C$ is relaxed: we have instead $x_i \geq 0 \quad \text{if } i \in I_C$.

A solution of CSCPP-PART$\left(I_C, \overline{I_C}\right)$ is a solution of CSCPP-EXT$\left(I_C, \overline{I_C}\right)$, because we simply removed the constraints $x_i > d_i^{1/\alpha}$ in the latter problem. Hence the execution time of the optimal solution of CSCPP-EXT$\left(I_C, \overline{I_C}\right)$ is lower than that of CSCPP-PART$\left(I_C, \overline{I_C}\right)$.

Furthermore, given a solution of CSCPP-EXT$\left(I_C, \overline{I_C}\right)$, one can easily see that its execution time in COSCHEDCACHE will be lower (the objective function is lower since it involves a minimum for all applications in I_C).

Lemma 2.4. *Given a set of applications $T_1, \ldots, T_n$ and a partition $I_C, \overline{I_C}$, the optimal solution to* CSCPP-EXT$\left(I_C, \overline{I_C}\right)$ *is*

$$x_i = \frac{(w_i f_i d_i)^{1/(\alpha+1)}}{\sum_{j\in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} \qquad \text{if } i \in I_C,$$
$$x_i = 0 \qquad \text{otherwise.}$$

Proof. We want to compute $\mathcal{X} = \{x_1, \ldots, x_n\}$ that minimizes the execution time. Discarding constant factors, this reduces to minimizing

$$K(\mathcal{X}) = \sum_{i\in I_C} \frac{w_i f_i d_i}{x_i^\alpha}$$

under the constraints: $x_i = 0$ if $i \in \overline{I_C}$, $x_i \geq 0$ otherwise, and $\sum_i x_i \leq 1$. Clearly, one can see that this last inequality is an equality when $I_C \neq \emptyset$ (otherwise K is not minimum).

To minimize the function, we compute the partial derivatives of K:

$$\forall i \in I_C,\ \frac{\partial K(\mathcal{X})}{\partial x_i} = -\alpha \frac{w_i f_i d_i}{x_i^{\alpha+1}}.$$

By setting them all to 0, we obtain the following equality for $1 \leq i \leq n$:

$$-\alpha \frac{w_i f_i d_i}{x_i^{\alpha+1}} = -\alpha \frac{w_n f_n d_n}{x_n^{\alpha+1}}.$$

Hence,

$$\forall i \in I_C,\ x_i = x_n \frac{(w_i f_i d_i)^{\frac{1}{\alpha+1}}}{(w_n f_n d_n)^{\frac{1}{\alpha+1}}};$$
$$\sum_{i=1}^{n} x_i = \frac{x_n}{(w_n f_n d_n)^{\frac{1}{\alpha+1}}} \sum_{i \in I_C} (w_i f_i d_i)^{\frac{1}{\alpha+1}}$$
$$= 1.$$

Hence, the desired result. □

Definition 2.4 (Dominant partition)**.** *Given a set of applications $T_1, \ldots, T_n$, we say that a partition of these applications $I_C, \overline{I_C}$ is* dominant*, if for all $i \in I_C$,*

$$\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{\sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} > d_i^{1/\alpha}.$$

We can now state the following result:

Theorem 2.2. *If a partition $I_C, \overline{I_C}$ is not dominant, then we can compute in polynomial time a better solution.*

Proof. Let $I_C, \overline{I_C}$ be a non-dominant partition.

Let $i_0 \in I_C$ such that $\frac{(w_{i_0} f_{i_0} d_{i_0})^{1/(\alpha+1)}}{\sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} \leq d_{i_0}^{1/\alpha}$.

First we can show that there is $i_1 \in I_C \setminus \{i_0\}$. Indeed, otherwise we would have $\frac{(w_{i_0} f_{i_0} d_{i_0})^{1/(\alpha+1)}}{\sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)}} = 1 \leq d_{i_0}^{1/\alpha}$, and $I_C, \overline{I_C}$ is not a valid partition: then CSCPP-PART$\left(I_C, \overline{I_C}\right)$ does not admit any solution.

Let $\mathcal{T}_e$ (resp. $\mathcal{T}_p$) be the optimal execution time of CSCPP-EXT$\left(I_C, \overline{I_C}\right)$ (resp. CSCPP-PART$\left(I_C, \overline{I_C}\right)$). We know that $\mathcal{T}_e \leq \mathcal{T}_p$. Let us further denote by $\mathcal{X} = \{x_1, \ldots, x_n\}$ the optimal solution to CSCPP-EXT$\left(I_C, \overline{I_C}\right)$. Let $\bar{\mathcal{X}} = \{\bar{x}_1, \ldots, \bar{x}_n\}$ be such that (i) $\bar{x}_{i_0} = 0$, (ii) $\bar{x}_{i_1} = x_{i_0} + x_{i_1}$, and (iii) $\bar{x}_i = x_i$ for all other i's.

Then clearly $\bar{\mathcal{X}}$ is a solution, and we have:

$$\begin{aligned}
&\mathcal{E}\text{xe}_{i_0}^{\text{seq}}(\bar{x}_{i_0}) \leq w_{i_0} \left(1 + f_{i_0} l_s + f_{i_0} l_l \frac{d_{i_0}}{x_{i_0}^{\alpha}}\right); \\
&\mathcal{E}\text{xe}_{i_1}^{\text{seq}}(\bar{x}_{i_1}) < w_{i_1} \left(1 + f_{i_1} l_s + f_{i_1} l_l \frac{d_{i_0}}{x_{i_1}^{\alpha}}\right); \qquad (2.8) \\
&\mathcal{E}\text{xe}_{i}^{\text{seq}}(\bar{x}_{i}) \leq w_i \left(1 + f_i l_s + f_i l_l \frac{d_i}{x_i^{\alpha}}\right) \qquad \text{if } i \in I_C; \\
&\mathcal{E}\text{xe}_{i}^{\text{seq}}(\bar{x}_{i}) = w_i \left(1 + f_i (l_s + l_l)\right) \qquad \text{if } i \in \overline{I_C}.
\end{aligned}$$

Indeed, these results are direct consequences of the definition of $\mathcal{E}\text{xe}^{\text{seq}}$, except Equation 2.8, which we establish as follows:

- If $x_{i_1} \geq d_{i_1}^{1/\alpha}$, then $\bar{x}_{i_1} > d_{i_1}^{1/\alpha}$

$$\begin{aligned}\mathcal{E}\text{xe}_{i_1}^{\text{seq}}(\bar{x}_{i_1}) &= w_{i_1}\left(1 + f_{i_1}l_s + f_{i_1}l_l\frac{d_{i_0}}{\bar{x}_{i_1}^{\alpha}}\right)\\ &< w_{i_1}\left(1 + f_{i_1}l_s + f_{i_1}l_l\frac{d_{i_0}}{x_{i_1}^{\alpha}}\right).\end{aligned}$$

- If $x_{i_1} < d_{i_1}^{1/\alpha}$, then for all $x \in [0,1]$, $\mathcal{E}\text{xe}_{i_1}^{\text{seq}}(x) < w_{i_1}\left(1 + f_{i_1}l_s + f_{i_1}l_l\frac{d_{i_0}}{x_{i_1}^{\alpha}}\right)$.

Hence:

$$\begin{aligned}\frac{1}{p}\sum_{i=1}^{n}\mathcal{E}\text{xe}_i^{\text{seq}}(\bar{x}_i) < \frac{1}{p}\Big(&\sum_{i\in\overline{I_C}} w_i(1 + f_i(l_s + l_l))\\ &+ \sum_{i\in I_C} w_i(1 + f_i l_s + f_i l_l\frac{d_i}{x_i^{\alpha}})\Big) = \mathcal{T}_e \leq \mathcal{T}_p,\end{aligned}$$

which shows that $\bar{\mathcal{X}}$ is a better solution computed in polynomial time from $\mathcal{X}$. Furthermore, by construction of $\bar{\mathcal{X}}$, we have strictly decreased the size of the new set I_C. □

We can show a second dominance result characterizing the optimal solution:

Theorem 2.3. *If a partition $I_C, \overline{I_C}$ is dominant, then the optimal solution to* CSCPP-PART$\left(I_C, \overline{I_C}\right)$ *is:*

$$\begin{aligned}x_i &= \frac{(w_i f_i d_i)^{1/(\alpha+1)}}{\sum_{j\in I_C}(w_j f_j d_j)^{1/(\alpha+1)}} && \text{if } i \in I_C;\\ x_i &= 0 && \text{otherwise.}\end{aligned}$$

Proof. This is a corollary of Lemma 2.4.

Indeed, this solution is the optimal solution to CSCPP-EXT$\left(I_C, \overline{I_C}\right)$ and it is a valid solution to CSCPP-PART$\left(I_C, \overline{I_C}\right)$, hence it is the optimal solution to CSCPP-PART$\left(I_C, \overline{I_C}\right)$. □

2.3.4 Extension of the dominance criterion for Amdahl applications

Finally, we provide extended definitions for non-perfectly parallel applications, by defining the dominant partition of both the parallel part and the sequential part of such applications.

Definition 2.5 (Dominant partition of parallel part)**.** *Given a set of applications $T_1, \ldots, T_n$, we say that a partition of these applications $I_C, \overline{I_C}$ is* dominant *for the parallel part if for all $i \in I_C$,*

$$\frac{(w_i f_i d_i(1 - s_i))^{1/(\alpha+1)}}{\sum_{j\in I_C}(w_j f_j d_j(1 - s_j))^{1/(\alpha+1)}} > d_i^{1/\alpha}.$$

Definition 2.6 (Dominant partition of sequential part)**.** *Given a set of applications $T_1, \ldots, T_n$, we say that a partition of these applications $I_C, \overline{I_C}$ is* dominant *for the sequential part if for all $i \in I_C$,*

$$\frac{(w_i f_i d_i s_i)^{1/(\alpha+1)}}{\sum_{j\in I_C}(w_j f_j d_j s_j)^{1/(\alpha+1)}} > d_i^{1/\alpha}.$$

The intuition behind these two definitions is the following: recall from Lemma 2.1 that the execution time is defined as $\mathcal{E}\text{xe}_i(p_i, x_i) = b_i + \frac{c_i}{p_i}$, with

$$A_i = 1 + f_i \left(l_s + l_l \cdot \min \left(1, \frac{m^i_{\text{1MB}\mathcal{S}_s}}{\left(\frac{x_i C_s}{10^6}\right)^{\alpha}} \right) \right),$$
$$b_i = A_i w_i s_i,$$
$$c_i = A_i w_i (1 - s_i).$$

We can observe that s_i, the sequential fraction, is key to decide which parts b_i or $\frac{c_i}{p_i}$ we should favor to minimize $\mathcal{E}\text{xe}_i(p_i, x_i)$. If $s_i << \frac{1}{p_i}$, then $\frac{c_i}{p_i}$ dominates the execution time, i.e., $\mathcal{E}\text{xe}_i(p_i, x_i) \approx c_i$. Hence the application could be seen as a perfectly parallel application where the new number of computing operations to do is $\tilde{w_i} = w_i(1 - s_i)$. Then Definition 2.5 is just a consequence of applying the definition of Dominant Partition to this new application.

Symmetrically, if s_i is large in front of one over the number of processors assigned to an application, then b_i dominates the execution time. Intuitively in this case, the number of processors by application is less important (and we will have a fair balance of processors). Hence, we want to favor applications with large values of $s_i w_i f_i d_i$. We verify these intuitions experimentally in Section 2.5.

2.4 Heuristics

In this section, we aim at designing efficient heuristics for general applications that obey Amdahl's law, and whose memory footprints are larger than the cache size ($a_i = +\infty$). However, the CoSched-Cache problem seems to be very difficult for such applications, as seen in Section 2.3.

We first explain how heuristics work, in particular to assign (rational numbers of) processors, in Section 2.4.1. The core of the heuristic consists in building a dominant partition, and we detail different possibilities to do so in Section 2.4.2. Finally, we propose a way to round the number of processors in case we need an integer number of processors, for instance if no multi-threading is allowed (see Section 2.4.3).

2.4.1 Structure of heuristics

We simplify the design of the heuristics by temporarily allocating processors as if the applications were perfectly parallel, and then concentrating on strategies that partition the cache efficiently among some applications (and give no cache fraction to remaining ones). In accordance with Theorem 2.2, our goal is to compute dominant partitions. Recall that I_C represents the subset of applications that receive a fraction of the cache. Once a dominant partition is given, we obtain the schedule $\mathcal{S} = \{(x_i, p_i)\}_i$ as follows: first we determine the x_i's with Theorem 2.3, and then we recompute the p_i's so that all applications complete simultaneously at time K. Indeed, while Lemma 2.2 does not hold for Amdahl applications, we still know thanks to Lemma 2.1 that all applications should complete simultaneously.

However, there is no longer a nice analytical characterization of the makespan K, hence we use a binary search to compute K as follows: for each application T_i, the execution time writes $(s_i + \frac{1-s_i}{p_i})c_i = K$, where s_i is the sequential fraction, and $c_i = w_i(1 + f_i(l_s + l_l \frac{d_i}{x_i^{\alpha}}))$ if $T_i \in I_C$, or $c_i = w_i(1 + f_i(l_s + l_l))$ otherwise. From $\sum_{i=1}^{n} p_i = p$, we derive the equation

$$\sum_{i=1}^{n} \frac{1 - s_i}{\frac{K}{c_i} - s_i} = p$$

Algorithm 1: DOM strategy, starting with all applications

```
1  procedure DOM (I, choice) begin
2      I_C ← I;
3      while ∃i ∈ I_C s.t. NOTDOM(i, I_C) do
4          k ← choice(I_C);
5          I_C ← I_C \ {k};
6          if I_C = ∅ then break;
7      end
8      Ī_C ← I \ I_C;
9      return (I_C, Ī_C);
10 end
```

Algorithm 2: DREV strategy, starting from empty set

```
1  procedure DREV (I, choice) begin
2      Ī_C ← I; I_C ← ∅;
3      k ← choice(Ī_C);
4      I'_C ← {k};
5      while ISDOM(I'_C) do
6          I_C ← I'_C;
7          Ī_C ← Ī_C \ {k};
8          if Ī_C = ∅ then break;
9          k ← choice(Ī_C);
10         I'_C ← I'_C ∪ {k};
11     end
12     return (I_C, Ī_C);
13 end
```

Figure 2.1: Two strategies to build dominant partitions.

and we compute K through a binary search. A lower (resp. upper) bound for K is to assign p (resp. 1) processor(s) to each application.

2.4.2 Computing a dominant partition

To compute dominant partitions, we use two greedy strategies:

- DOM: we start with $I_C = \mathcal{I}$ and greedily remove some applications from I_C until we have a dominant partition (see Algorithm 1); NOTDOM(i, I_C) returns true if i does not satisfy the definition of dominant partition for I_C;
- DREV: initially I_C is empty, and we greedily add applications while I_C remains dominant (see Algorithm 2); ISDOM(I'_C) returns true if I'_C is a dominant partition.

Both strategies come in three flavors, depending on the dominance definition that we use. From Definition 2.4, we get that NOTDOM(i, I_C) is true if and only if

$$\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \leq \sum_{j \in I_C} (w_j f_j d_j)^{1/(\alpha+1)},$$

and ISDOM(I'_C) is true if and only if

$$\forall i \in I'_C, \frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} > \sum_{j \in I'_C} (w_j f_j d_j)^{1/(\alpha+1)},$$

for strategies DOM and DREV. If we use Definition 2.6, we simply replace all w_k's by $w_k s_k$ (strategies DOMS and DREVS focusing on the sequential part), while with Definition 2.5, we replace all w_k's by $w_k(1 - s_k)$ (strategies DOMP and DREVP focusing on the parallel part).

For each of these strategies, the greedy criterion to select the next application is the *choice* function taken from the following three alternatives:

- RANDOM: $choice(\mathcal{I})$ picks up randomly one application among all applications;
- MINRATIO considers the ratio that appears in Definition 2.4, Definition 2.6 or Definition 2.5 (dominant partitions), and chooses an application with a small ratio; for DOM and DREV, we have:
$$choice(\mathcal{I}) = \arg\min_{i\in\mathcal{I}} \left(\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \right);$$
and we replace w_i by $w_i s_i$ in DOMS and DREVS, or by $w_i(1-s_i)$ in DOMP and DREVP;
- MAXRATIO proceeds the other way round, by choosing an application with a large ratio, simply replacing the $\arg\min$ by an $\arg\max$.

The intuition behind these heuristics is the following: applications that make the solution non dominant for DOM and DREV are such that (see Definition 2.4):

$$\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}} \leq \sum_{j\in I_C} (w_j f_j d_j)^{1/(\alpha+1)}.$$

Hence, we expect to reach dominance faster by removing from a non-dominant solution applications with low $\frac{(w_i f_i d_i)^{1/(\alpha+1)}}{d_i^{1/\alpha}}$ (left term of the equation). Intuitively, DOM, DOMS and DOMP should work well with the MINRATIO criterion. For symmetric reasons, we expect DREV, DREVS and DREVP to work well with the MAXRATIO criterion. These intuitions will be experimentally confirmed in Section 2.5.

Altogether, by combining six strategies, and with three different *choice* functions for each strategy, we obtain 18 heuristics to build dominant partitions. We denote by DOM-MINRATIO the DOM strategy using MINRATIO as a *choice* function, and we use a similar notation for all heuristics.

2.4.3 Integer processor assignment

Based on the rational cache allocation, we want to give an integer processor allocation in order to tackle architectures that do not allow to share processors between applications through multi-threading. The choice functions above are first used to build a dominant partition, then we assign cache based on that partition to obtain the x_i's. In Algorithm 3, the set $\mathcal{I}$ contains all applications and x is the set that contains all x_i's. Finally, p is the total number of processors and n the total number of applications (i.e., $n = |\mathcal{I}|$). After the cache is assigned, we initialize processor assignment by giving one processor to each application, and the remaining processors are assigned in a greedy way: assign one processor to the application currently with longest execution time, until all processors are assigned. It should be noted that integer processor assignment will only work when $p \geq n$, since each application needs at least one processor.

2.5 Simulations

To assess the efficiency of the heuristics defined in Section 2.4, we have performed extensive simulations. The simulation settings are discussed in Section 2.5.1, and results are presented in Section 2.5.2 (comparison of the 18 heuristics of Section 2.4), Section 2.5.3 (assessing the gain due to co-scheduling), and Section 2.5.4 (with integer numbers of processors). The code is publicly available at `http://perso.ens-lyon.fr/loic.pottier/archives/cache-int.zip`.

Algorithm 3: Integer processor assignment

```
procedure IntegerProcessor (x, p, I)
begin
    for i ∈ I do p'_i = 1;
    p_remain = p − n;
    while p_remain > 0 do
        i = arg max_{k∈I} (Exe_k(p'_k, x_k));
        p'_i = p'_i + 1;
        p_remain = p_remain − 1;
    end
    return p'_i;
end
```

2.5.1 Simulation settings

We use data from applicative benchmarks to run the experiments. Table I provides a brief description of the NAS Parallel Benchmark (NPB) suite [11], and shows the parameters for these six HPC applications.

App$_i$	Description	w_i	f_i	$m^i_{40\text{MB}\mathcal{S}_s}$
CG	Uses conjugate gradients method to solve a large sparse symmetric positive definite system of linear equations	5.70E+10	5.35E-01	6.59E-04
BT	Solves multiple, independent systems of block tridiagonal equations with a predefined block size	2.10E+11	8.29E-01	7.31E-03
LU	Solves regular sparse upper and lower triangular systems	1.52E+11	7.50E-01	1.51E-03
SP	Solves multiple, independent systems of scalar pentadiagonal equations	1.38E+11	7.62E-01	1.51E-02
MG	Performs a multi-grid solve on a sequence of meshes	1.23E+10	5.40E-01	2.62E-02
FT	Performs discrete 3D fast Fourier Transform	1.65E+10	5.82E-01	1.78E-02

Table I: Description and experimental values from NPB benchmarks.

We obtain the values shown in Table I by instrumenting and simulating the benchmarks (*CLASS=A*) on 16 cores using PEBIL [72]. For the simulations, we use a cache configuration representing an Intel Xeon CPU E5-2690, with a 40MB last level cache per processor of 8 cores. Since the cache miss ratio is defined for a 40MB cache, we have $d_i = m^i_{40\text{MB}\mathcal{S}_s} \left(\frac{40\times10^6}{C_s}\right)^{\alpha}$.

We consider three sets of data for simulations:

- NPB-6: Limited to the six applications defined in Table I;
- NPB-SYNTH: We build synthetic applications from Table I with only varying randomly the work w_i between 1E+8 and 1E+12;

- RANDOM: We build synthetic applications from Table I with varying all values randomly. The work w_i is taken between 1E+8 and 1E+12, f_i between 1E-01 and 9E-01, and $m^i_{40\text{MB}\mathcal{S}_s}$ between 1E-02 and 9E-04.

The sequential fraction of work s_i is taken randomly between 1% and 15%.

For the execution platform, we consider one many-core *Sunway TaihuLight* [35] with 256 processors and a shared memory of 32GB. We chose this platform because of its high core count. Strictly speaking, this platform does not have a last level cache (LLC), but the shared memory can be seen as the LLC, using the disk as the large memory. We have $C_s = 32 \times 10^9$. The large storage latency l_l is set to 1. The small storage latency l_s is set to 0.17. According to the literature [69, 82, 95], the last level cache (LLC) latency is on average four to ten times better than the DDR latency, and we enforce a ratio of 5.88 in the simulations. We have used different ratios and they lead to similar results (see Figure 2.13). Finally, the Power Law parameter is set to $\alpha = 0.5$. We execute each heuristic 50 times and we compute the average *makespan*, i.e., the longest execution time among all co-scheduled applications.

2.5.2 Comparison of the heuristics

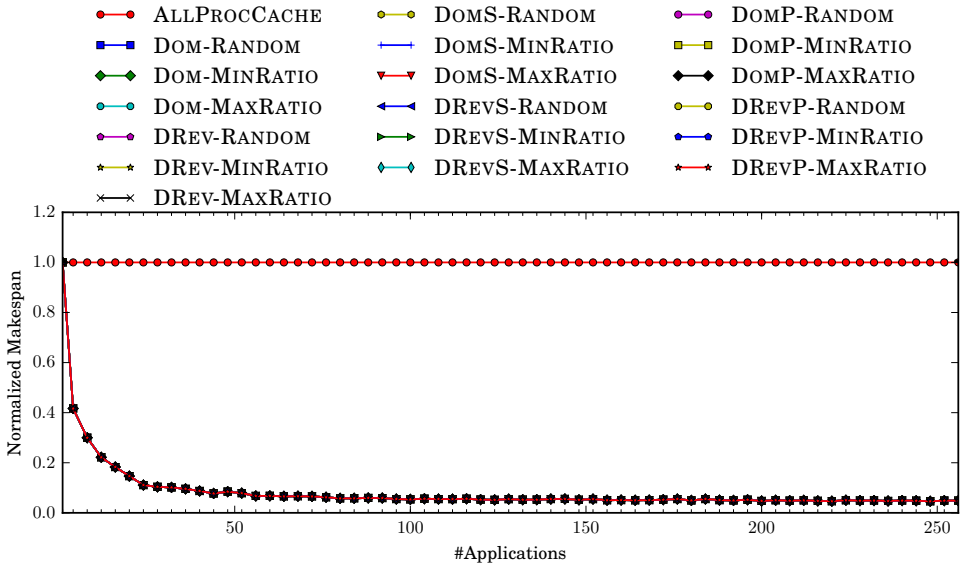
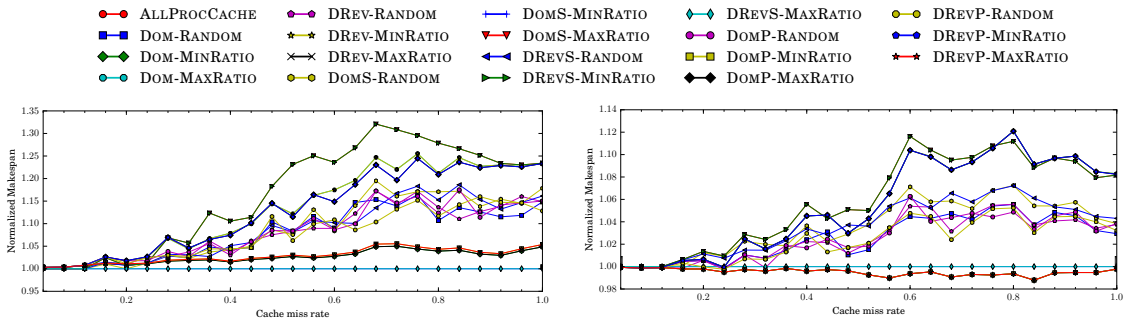


Figure 2.2: Comparison of all dominant partition heuristics on 256 processors with NPB-SYNTH.

Figure 2.2 shows the normalized makespan obtained by all of the heuristics building dominant partitions. We set the number of processors to 256. Results are normalized with the makespan of AllProcCache, which is the execution without any co-scheduling: in the AllProcCache heuristic, applications are executed sequentially, each using all processors and all the cache. We vary the number of applications between 1 and 256. The eighteen heuristics obtain similarly good results, with a gain of 85% over AllProcCache as soon as there are at least 50 applications.

Since all eighteen variants show the same performance on the previous data sets, we investigate the impact of the cache miss rate by varying it between 0 and 1 with a LLC of $C_s = 1GB$ in Figure 2.3. Results are now normalized with DomS-MinRatio in both figures, which enables to zoom out the differences.

The first noticeable result from Figure 2.3 is that for all versions of the strategies that build dominant strategies, MinRatio performs better with strategies that remove applications from the I_C (Dom,



(a) s_i randomly set between 0.01 and 0.15.

(b) s_i randomly set between 0.001 and 0.01.

Figure 2.3: Impact of the cache miss ratio $m^i_{40\text{MB}\mathcal{S}_s}$ with a 1GB cache and 16 applications with NPB-SYNTH.

DomS, DomP), whereas MaxRatio works better with strategies that add applications to the I_C (DRev, DRevS, DRevP). This confirms the mathematical intuition presented in Section 2.4.

Furthermore, we confirm the mathematical intuition on the influence of the Amdahl factor (s_i) presented in Section 2.3.4:

- We observe that in Figure 2.3a, when the sequential fraction is not negligible (s_i chosen uniformly at random between 0.01 and 0.15), DomS-MinRatio and DRevS-MaxRatio are always the best (their plots overlap), with a gain from 10 to 15% with respect to the random-based heuristics when the cache miss rate is greater than 0.5.
- On the contrary, when it is negligible (s_i chosen uniformly at random between 0.001 and 0.01), then the DomP-MinRatio and DRevP-MaxRatio versions perform better.

Note that overall, the observable differences between heuristics is mainly when the cache miss ratio is large. According to current data, $m_{40\text{MB}\mathcal{S}_s}$ ranges from 1E-02 to 1E-04 (see Table I). In addition, these differences are visible only with a small shared memory (1GB in the example), while our execution platform has a 32GB shared memory. Overall, for the system used in these simulations, all heuristics perform similarly, even though DomS-MinRatio and DRevS-MaxRatio seem to perform best in all other settings that we tried

In the following simulations, the sequential fraction will always, unless otherwise mentioned, be taken between 1% and 15%. Therefore, for clarity, we plot only one heuristic based on dominant partitions in the remaining simulations, namely DomS-MinRatio.

2.5.3 Gain with co-scheduling

In this section, we assess the gain due to co-scheduling by comparing DomS-MinRatio with AllProcCache and with three other heuristics:

- Fair gives $p_i = \frac{p}{n}$ processors, and a fraction of cache $x_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$ to each application;
- 0Cache gives no cache to any application, i.e., $x_i = 0$ for $1 \leq i \leq n$, and then it computes the p_i's so that all applications finish at the same time;
- RandomPart randomly partitions applications with and without cache. For those in cache, the x_i's are computed with the method used for dominant partitions. Then, the p_i's are computed so that all applications finish at the same time.

Impact of the number of applications

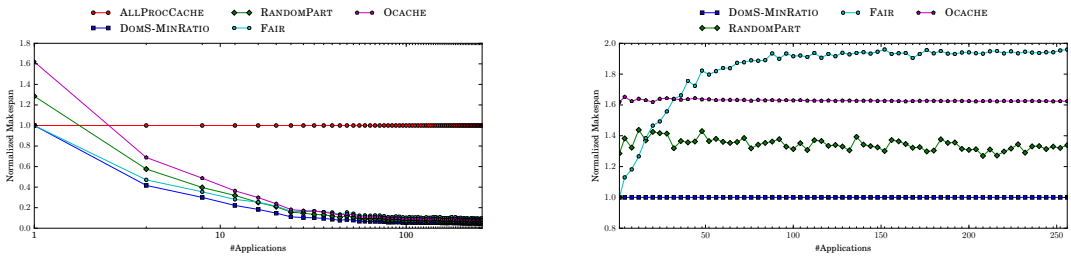


Figure 2.4: Impact of the number of applications with NPB-SYNTH.

Figure 2.4 (normalized with AllProcCache on the left) shows the impact of the number of applications when the number of processors is set to 256 with NPB-SYNTH. We see that DomS-MinRatio outperforms the other heuristics, hence showing the efficiency of our approach based on dominant partitions. Results are also normalized with DomS-MinRatio (on the right), so that we can better observe the differences between co-scheduling heuristics. Fair exhibits good results only for a small number of applications, when all applications can fit into cache. Otherwise, the use of dominant partitions is much more efficient, as seen with RandomPart, or even 0cache that does not use cache but ensures that all applications finish at the same time. These results show the accuracy of the model and the benefits of using dominant partitions. Also, we note the importance of cache partitioning, since the difference between 0cache and DomS-MinRatio relies on cache allocation. Figure 2.5 (normalized with AllProcCache and DomS-MinRatio) shows the impact of the number of applications when the number of processors is set to 256 with RANDOM. We observe similar results with RANDOM and NPB-SYNTH. Dominant partition heuristics still outperform other heuristics.

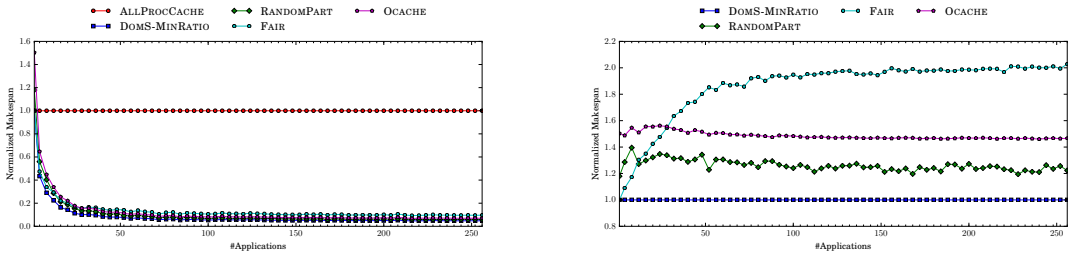


Figure 2.5: Impact of the number of applications with RANDOM.

Impact of the number of processors

Figure 2.6 (normalized with AllProcCache on the left) shows the impact of the number of processors when the number of applications is set to 16. When the number of processors increases, the gain of co-scheduling increases. In both figures, DomS-MinRatio and outperforms other methods. RandomPart, which builds a random partition instead of a dominant one, is outperformed by DomS-MinRatio, and the latter is the only heuristic that surpasses AllProcCache when the number of processors is low. So, building a dominant partition seems a good strategy to optimize the makespan.

The normalization with DOMS-MINRATIO (on the right) shows that when the number of processors increases, FAIR becomes better, while RANDOMPART and 0CACHE are quite stable since they are based on the same model as DOMS-MINRATIO. The only difference between 0CACHE and DOMS-MINRATIO is the cache allocation strategy, and the gain from cleverly distributing cache fractions across applications exceeds 20%. With more applications, we obtain the same ranking of heuristics, except that FAIR is always the worst heuristic: since there are less processors on average per application, a good co-scheduling policy is necessary.

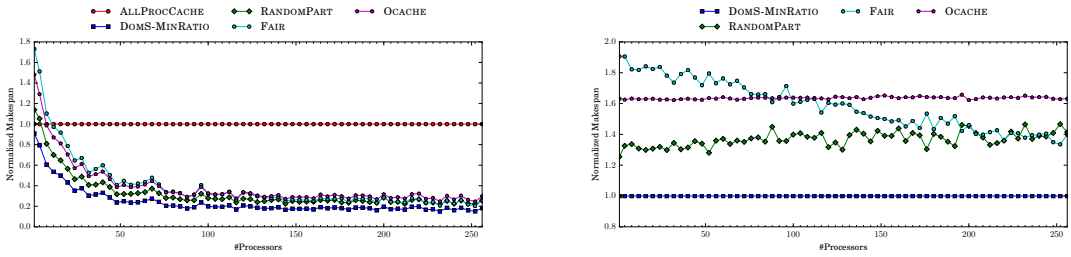


Figure 2.6: Impact of the number of processors with NPB-SYNTH.

Figure 2.7 (normalized with ALLPROCCACHE and DOMS-MINRATIO) shows the impact of the number of processors with NPB-6. The number of applications is set to 6. We observe with less applications that FAIR obtains better results than 0CACHE when the number of processors is bigger than 50.

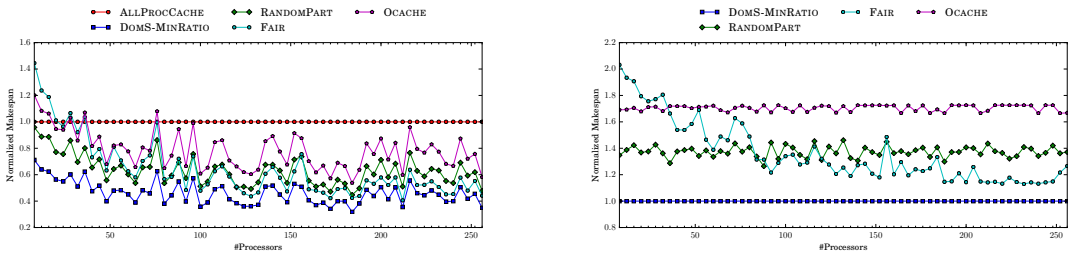


Figure 2.7: Impact of the number of processors with NPB-6.

Figure 2.8 (normalized with ALLPROCCACHE and DOMS-MINRATIO) shows the impact of the number of processors with RANDOM. The number of applications is set to 16. We obtain similar results with RANDOM and NPB-SYNTH.

Figure 2.9a (normalized with DOMS-MINRATIO) shows the impact of the number of processors with 64 applications. Compared to Figure 2.6, the main difference is that FAIR now obtains the worst performance, even 0CACHE is better. This difference in performance for FAIR is due to a higher number of applications. As each application receive a fraction of cache and a fraction of processors, each of them obtains less resources when the number of applications increases. Figure 2.9b (normalized with ALLPROCCACHE and DOMS-MINRATIO) shows the impact of the number of processors with RANDOM and 64 applications. As expected, we obtain similar results, 0CACHE and RANDOMPART show better performance when the number of applications increases. DOMS-MINRATIO is still the best heuristic, the number of processors does not affect relative performance.

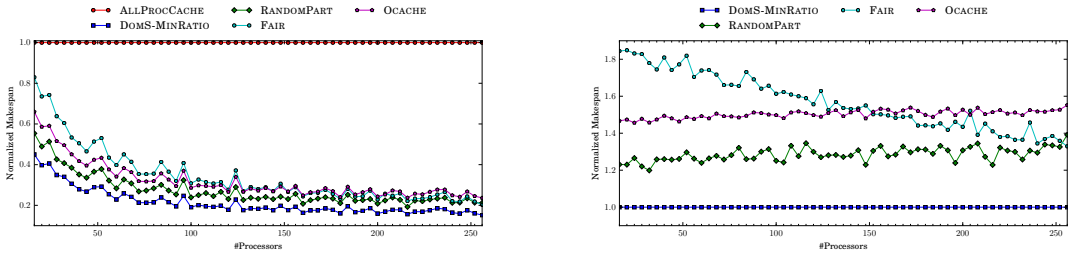
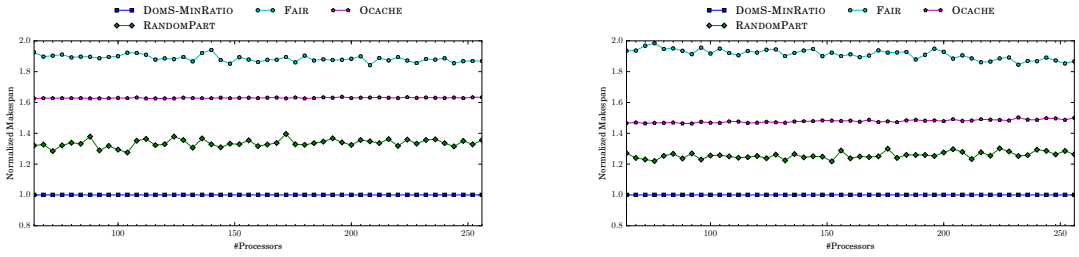


Figure 2.8: Impact of the number of processors with RANDOM.



(a) NPB-SYNTH (b) RANDOM

Figure 2.9: Impact of the number of processors with 64 applications (normalized with DOMS-MINRATIO)

Impact of the sequential fraction of work

Figure 2.10 (normalized with ALLPROCCACHE) shows the impact of the sequential part s_i when the number of processors is set to 256. The number of applications is set to 16. As expected, when the sequential fraction of work increases, all co-scheduling heuristics perform better than ALLPROCCACHE, and DOMS-MINRATIO is always the best heuristic. It leads to a gain of more than 50% when $s_i = 0.01$. The normalization with DOMS-MINRATIO better shows the impact of the sequential part: we observe that when the sequential fraction of work increases, FAIR obtains results closer to DOMS-MINRATIO.

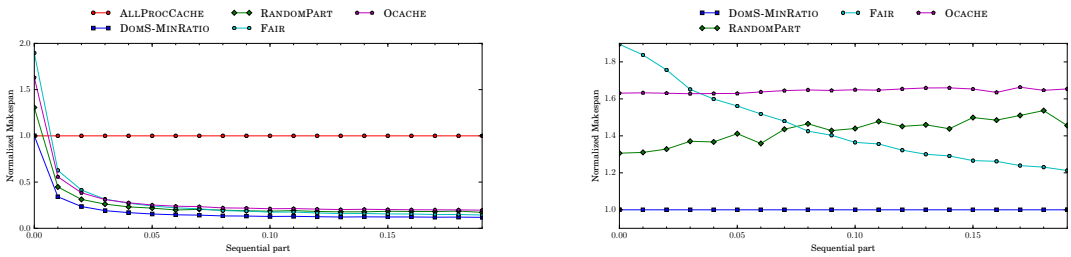


Figure 2.10: Impact of sequential fraction of work with NPB-SYNTH.

Figure 2.11 (normalized with ALLPROCCACHE and DOMS-MINRATIO) shows the impact of the sequential fraction of work with NPB-6 (6 applications). We observe that the performance of FAIR

increases when the sequential fraction of work increases. Indeed, more the sequential fraction of work is important, more the cache allocation becomes crucial.

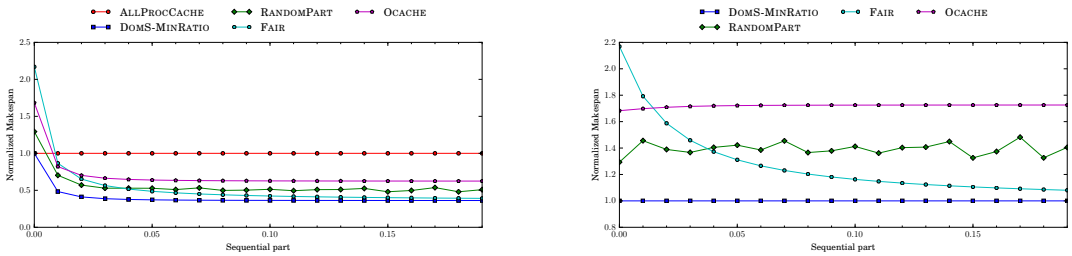


Figure 2.11: Impact of sequential fraction of work with NPB-6.

Figure 2.12 (normalized with AllProcCache and DomS-MinRatio) shows the impact of the sequential fraction of work with RANDOM and 16 applications. We observe similar results to the previous one obtained with NPB-SYNTH.

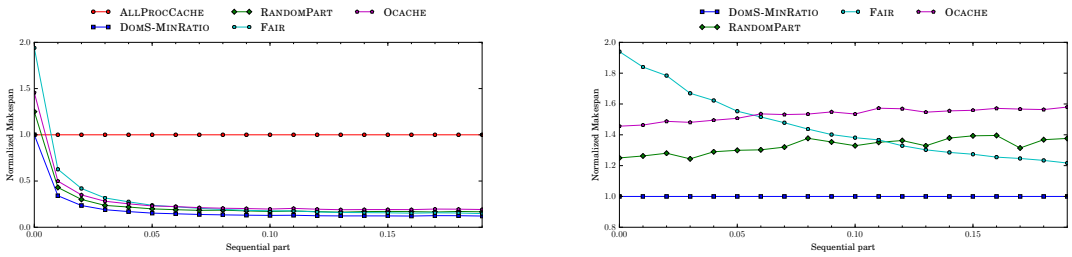


Figure 2.12: Impact of sequential fraction of work with RANDOM.

Impact of the cache latency

Figure 2.13 (normalized with AllProcCache) shows the impact of the cache latency l_s with NPB-SYNTH and 16 applications (on the left) on 256 processors. The sequential fraction of work is set to $s_i = 0.0001$ for all i. We observe that the l_s cost does not have an impact on relative performance. Right side of Figure 2.13 (normalized with AllProcCache) shows the impact of the cache latency l_s with NPB-SYNTH and 64 applications on 256 processors. The sequential fraction of work is set to $s_i = 0.0001$ for all i. As on the previous figure, we see that the l_s cost does not have an impact of relative performance, even with 64 applications.

Impact of the cache miss rate

Figure 2.14 (normalized with DomS-MinRatio) shows the impact of the cache miss rate with NPB-SYNTH and 16 applications. We vary the cache miss rate $m^i_{40\text{MB}\mathcal{S}_s}$ between 0 and 1. When the cache miss rate increases, the performance of RandomPart and 0Cache increases. Indeed, when the rate of miss increases, using the cache is less important, so 0Cache becomes competitive. But, we have to keep in mind that, with real applications, the cache miss rate rarely exceeds 20%.

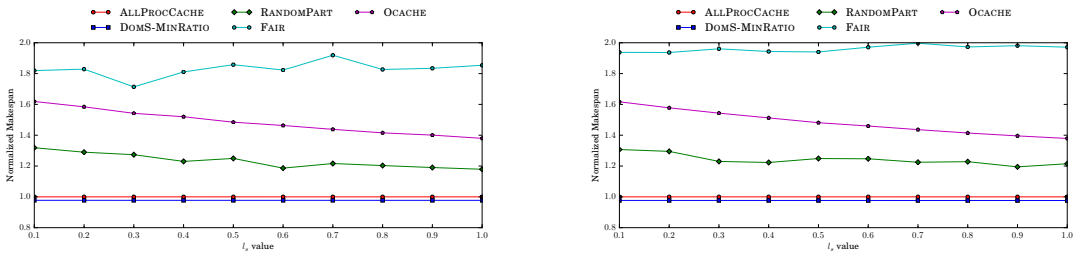


Figure 2.13: Impact of latency l_s with NPB-SYNTH with 16 and 64 applications.

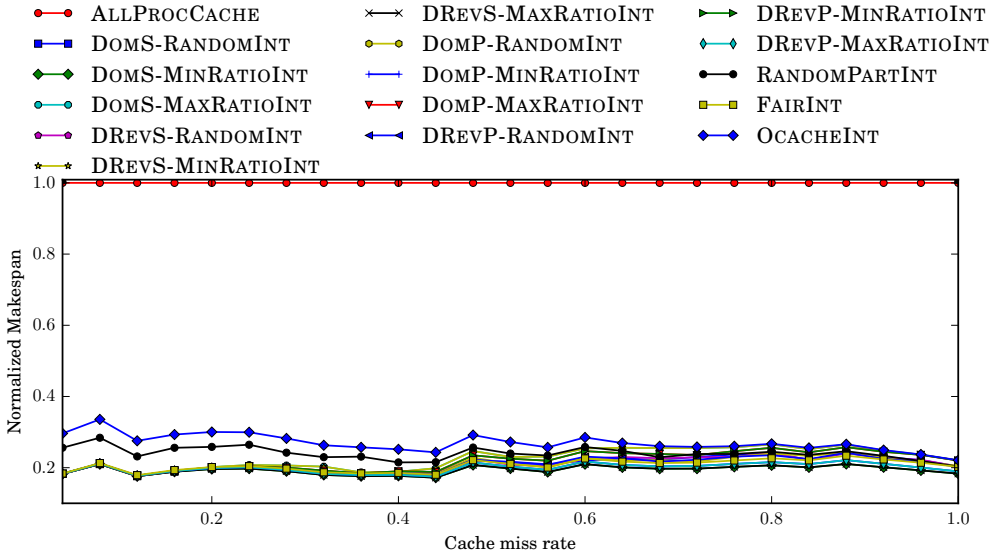


Figure 2.14: Impact of cache miss rate using a 1GB LLC.

Processor and cache repartition

Figure 2.15 shows the processor repartition and cache repartition when we vary the number of applications from 1 to 256 with 256 processors with NPB-SYNTH. We use an error bar plot where the error interval represents here the maximum and minimum number of processors (or cache fraction) allocated to an application. As expected, we observe that the range between minimum and maximum decreases when the number of applications increases. The processor allocation of FAIR is not interesting, the maximum is always equal to the minimum because we allocate the same number of processors to each application.

Since all dominant partition heuristics give the same results, we only use DOMS-MINRATIO. The repartition of processors for 0CACHE is interesting: it turns out to be very close to the repartition obtained with DOMS-MINRATIO, even though it is not using cache.

Figure 2.16 shows the processor repartition and cache repartition when we vary the number of applications from 1 to 256 with 256 processors with RANDOM. The results with RANDOM are very similar to the results obtained with NPB-SYNTH. However, note that cache allocation with FAIR is more heterogeneous when we have random application profiles.

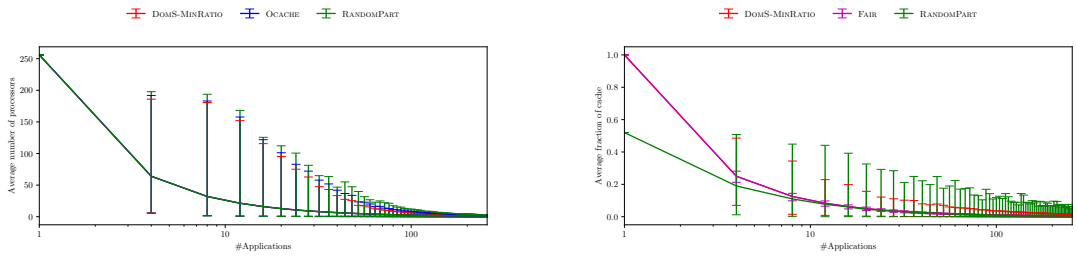


Figure 2.15: Processor and cache repartition with 256 processors with NPB-SYNTH.

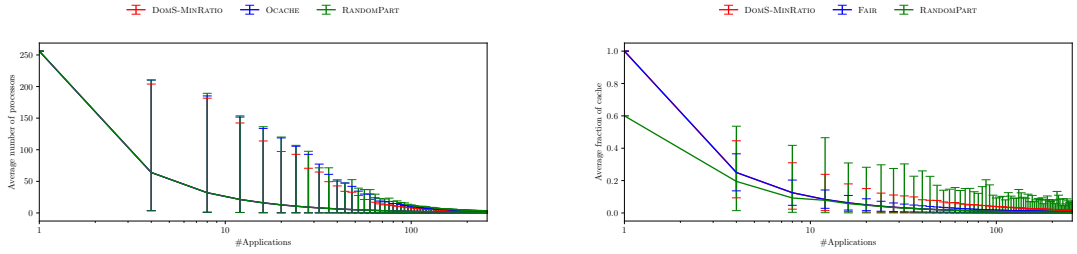


Figure 2.16: Processor and cache repartition with 256 processors with RANDOM.

Summary

To summarize, all heuristics based on dominant partitions are very efficient, especially when compared to the classical heuristics FAIR (which shares the cache fairly between applications) and ALLPROCCACHE (which does no co-scheduling). The unexpected result that can be observed is that the gain brought by our heuristics comes even with very low sequential time (below 0.01)! This is unexpected since the natural intuition would be a behavior such as the one observed on FAIR: a makespan up to 1.9 times longer than ALLPROCCACHE with low sequential time.

We show that the ratio processors/applications has a significant impact on performance: when many processors are available for a few applications, it is less crucial to use efficient cache-partitioning and all applications can share the cache, hence FAIR obtains good results, close to DOMS-MINRATIO. Otherwise, RANDOMPART is the second best heuristic. A surprising information that also confirms the strength of our partition based heuristics is that *natural* heuristics such as FAIR and ALLPROCCACHE perform worse than 0CACHE our implementation with no usage of cache.

All heuristics run within a very small time (less than ten seconds in the worst of the settings used, to be compared with a typical application execution time in hours or days), hence they can be used in practice with a very light overhead.

2.5.4 With an integer number of processors

In this section, we study the impact of rounding the number of processors to an integer number on heuristics. We focus again mainly on DOMS-MINRATIO, and we add the suffix INT to heuristic names to denote the fact that we use Algorithm 3 to compute an integer processor allocation.

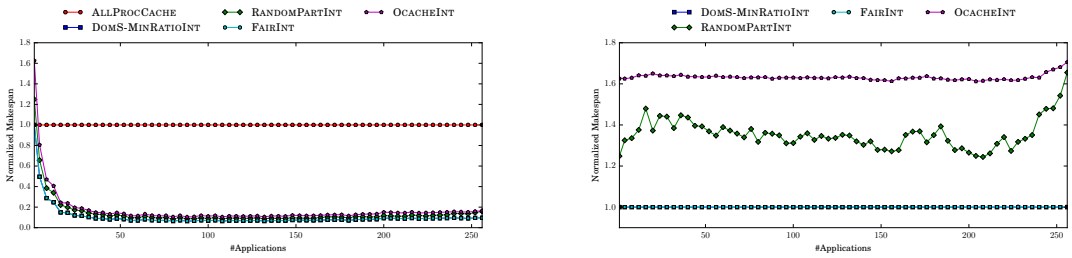


Figure 2.17: Impact of the number of applications with NPB-SYNTH.

Impact of the number of applications

In this simulation, we vary the number of applications from 1 to 256 on 256 processors. Figure 2.17 is normalized with AllProcCache (on the left), and heuristics obtain a similar relative performance as in Section 2.5.3, with a gain of 90% over AllProcCache as soon as there are at least 50 applications. The right side of Figure 2.17 shows the performance of the same heuristics but normalized with DomS-MinRatioInt. As expected, 0CacheInt is the worst, and RandomPartInt performs always in the middle between 0CacheInt and FairInt. As we use the same algorithm to round the rational processor allocation, the differences in performance mostly rely on cache allocation.

The fact that FairInt and DomS-MinRatioInt give similar results show that the cache allocation of DomS-MinRatioInt must not be far from the fair distribution of FairInt. However, contrarily to Fair, processors are not equally shared between applications but distributed according to their needs, hence the much better performance of FairInt compared to Fair.

Simulations showing the impact of the number of processors and of the sequential fraction of work give similar results, with FairInt and DomS-MinRatioInt overlapping and beating other heuristics.

Impact of the number of processors

Figure 2.18 shows the impact of the number of processors when the number of application is set to 16 and the number of processor very between 16 and 256. The left figure is normalized with AllProcCache and the right figure is normalized with DomS-MinRatioInt. As for previous results, all heuristics outperform AllProcCache, the performance of heuristic methods does not get better with the growth of processor number when the processor number get bigger than 24. However, all heuristics obtain a gain of 60% on average. The right figure helps us to zoom on details, DomS-MinRatioInt and FairInt are overlapping. All heuristics get better with the increasing of the processor number, and perform almost as good as DomS-MinRatioInt and FairInt when the number of processors reach 100. From Figure 2.17, we can find out that average number of processors per application is one of the most critical parameter to obtain good performance.

Impact of the sequential fraction and the cache miss rate

As DomS-MinRatioInt and FairInt show the same performance, we study the impact of the sequential fraction and the cache miss rate, as we did in Section 2.5.2 (Figure 2.19). The number of applications is set to 16 and the number of processors to 256 with a LLC of $C_s = 1GB$. The results are normalized with DomS-MinRatioInt. On the left side of Figure 2.19, we compare all dominant partition heuristics by varying the sequential fraction when the cache miss rate is set to 0.8 in order

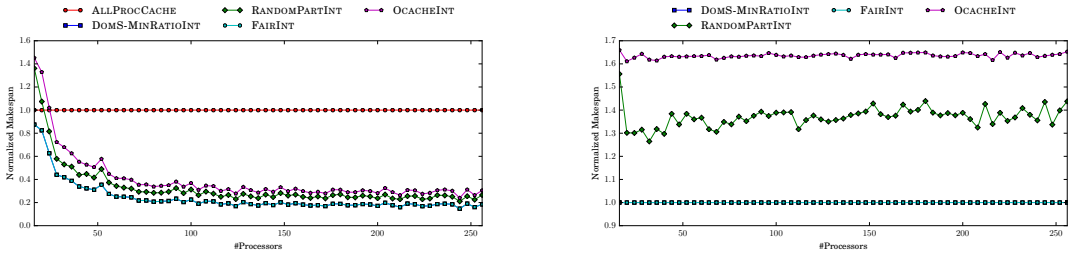


Figure 2.18: Impact of the number of processors.

to see differences between heuristics. We note that the dominant partition heuristics favoring the sequential part outperform the others, especially the ones favoring the parallel part. DOM-MINRATIOINT and DREV-MAXRATIOINT overlap with DOMS-MINRATIOINT. All variants using RANDOM criterion perform on average around 1.10. As expected, giving more cache to applications with bigger sequential fractions is better. In the right figure, we vary the cache miss rate between 0 and 1 This figure is interesting due to the difference of performance between DOMS-MINRATIOINT and FAIRINT. Clearly, the difference of performance between heuristics when we use integer processors rely on cache allocation. When the cache miss ratio increases, the performance of DOMS-MINRATIOINT becomes better. When the cache miss rate is larger than 0.01, DOMS-MINRATIOINT outperforms all other heuristics, and we obtain an average gain of 10% on FAIRINT. The performance of 0CACHEINT becomes better when the cache miss rate increases.

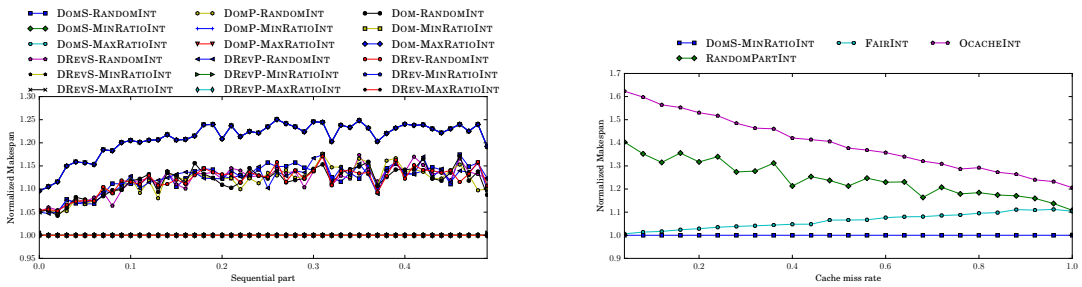


Figure 2.19: Impact of the sequential fraction and the cache miss rate with NPB-SYNTH.

Figure 2.20 shows the performance obtained when the sequential fraction of work vary. The number of applications is set to 16 and the number of processor is set to 256. The left figure is normalized with ALLPROCCACHE and the right one is normalized with DOMS-MINRATIOINT. We can see from both figures that DOMS-MINRATIOINT and FAIRINT overlaps, and both of them outperform other heuristic methods.

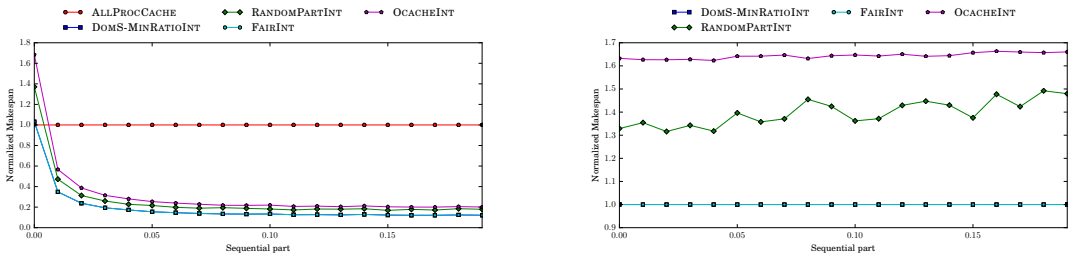


Figure 2.20: Impact of sequential fraction with NPB-SYNTH.

Summary

To summarize, when we use integer processors, all heuristics based on dominant partitions are still very efficient, but those that favor either the sequential part or none of them perform better. The main difference between results with rational and integer processor assignments is that DOMS-MINRATIOINT and FAIRINT overlap if the cache miss rate is low (less than 1%), because of the better processor assignment for FAIRINT. We show that the cache miss rate has a significant impact on performance: when many cache misses occur, it is more crucial to use efficient cache-partitioning and all applications can share the cache, hence DOMS-MINRATIOINT outperforms FAIRINT when the cache miss rate is larger than 10%. As expected, DOMS-MINRATIOINT performs better when the cache miss rate increases. Otherwise, RANDOMPARTINT is the third best heuristic, followed by 0CACHEINT that does not use the cache.

2.6 Conclusion

In this chapter, we have provided a preliminary study on co-scheduling algorithms for cache-partitioned systems, building upon a theoretical study. The two key scheduling questions are (i) which proportion of cache and (ii) how many processors should be given to each application. For rational numbers of processors, we proved that the problem is NP-complete, but we have been able to characterize optimal solutions for perfectly parallel applications by introducing the concept of *dominant partitions*: for such applications, we have computed the optimal proportion of cache to give to each application in the partition. Furthermore, we have provided explicit formulas to express the number of processors to assign to each application.

Several polynomial-time heuristics focusing on Amdahl's applications have been built upon these results, both for rational and integer numbers of processors. Extensive simulation results demonstrate that the use of dominant partitions always leads to better results than more naive approaches, as soon as there is a small sequential fraction of work in application speedup profiles. The concept of sharing the cache only between a subset of applications seems highly relevant, since even an approach with a random selection of applications that share the cache leads to good results. Also, a clever partitioning of the cache pays off quite well, since our heuristics lead to a significant gain compared to an approach where no cache is given to applications. Overall, the heuristics appear to be very useful for general applications, even though their cache allocation strategy rely mainly on simulating a perfectly parallel profile.

For future work, on the theoretical side, we plan to focus on the problem with integer numbers of processors and we hope to derive interesting results that could help design even more efficient heuristics. On the practical side, Chapter 3 presents real experiments done on a cache-partitioned system with a high core count, hence validating the accuracy of the model and confirming the impact of these promising results.

Chapter 3

Co-scheduling HPC workloads on cache-partitioned CMP platforms

Based on the results obtained in Chapter 2, we pursue the study of co-scheduling algorithms with cache partitioning techniques but, this time, using a real cache-partitioned multiprocessor (Intel Xeon) to assess the interest of cache partitioning on such platforms. Intel recently introduced a new hardware feature for cache partitioning called *Cache Allocation Technology* (CAT) [87]. CAT allows the programmer to reserve cache subsections, so that when several applications execute concurrently, each of them has its own cache area. Using CAT, Lo et al. [76] showed experimentally that important gains could be reached by co-scheduling latency-sensitive applications with a strict cache partitioning.

In this chapter, we also use CAT to partition the LLC into several areas when co-scheduling applications, but with the objective of optimizing the throughput of *in-situ* or *in-transit* analysis for large-scale simulations. Indeed, in such simulations, data is generated at each iteration and periodically analyzed by parallel processes on dedicated nodes, concurrently of the main simulation [104]. If these dedicated nodes belong to the main simulation platform (thereby reducing the number of available cores for simulation), we speak of *in-situ* processing, while if they belong to an auxiliary platform, we speak of of *in-transit* processing [13]. In both cases, several applications (various kernels for analysis) have to run concurrently to analyze the data in parallel of the current simulation step. The constraint is to achieve a prescribed throughput for each application, because the outcome of the analysis drives the next steps of the simulation. In the simplest case, each application will have to complete within the time of a simulation step, hence we need to achieve the same throughput for each application, and maximize that value. In other situations, some applications may be needed only every k simulation steps, with a different value of k per application [77]. This calls for achieving a weighted throughput per application, and for maximizing the minimum value of these weighted throughputs, which dictates the global rate at which the analysis can progress.

Note that in Chapter 2, we were only considering the makespan of the co-schedule, while we aim here at maximizing a weighted throughput. Indeed, this new objective better fits the target applications that we execute on the platform. A second difference, besides doing actual experiments, is to specialize our study on iterative HPC kernels, instead of general applications obeying Amdahl's law as in Chapter 2. Finally, we focus exclusively on integer numbers of cache fractions and processors, since fractions cannot be assigned on the Intel Xeon.

Main contributions. The first major contribution of this chapter is to introduce a model that characterizes application performance. Next, we provide strategies to decide how many cores and which cache fraction should be assigned to each application, in order to maximize the weighted throughput. A

dynamic programming algorithm provides an optimal strategy, according to the model. The last major contribution is to provide an extensive set of experiments conducted on the Intel Xeon, which assesses the gains achieved by our optimal resource allocation strategy. We therefore demonstrate that cache-partitioning strategies can lead to gains in performance for in-situ analysis for large-scale simulations.

The rest of the chapter is organized as follows. Section 3.1 details the main framework and all application/platform parameters, as well as the optimization problem. Section 3.2 presents five co-scheduling strategies, including a dynamic programming approach that provides an optimal resource assignment (according to the model). Section 3.3 describes the real cache partitioned platform used to perform the experiments. Section 3.4 assesses the accuracy of the model. Section 3.5 reports extensive experiments. Finally, Section 3.6 summarizes our main contributions and discusses directions for future work. A review of the related work on co-scheduling and cache partitioning techniques can be found in Chapter 2, Section 2.1.

3.1 Model and optimization problem

The objective is to execute m iterative applications $A_1, \ldots, A_m$ on P identical cores. The applications are sharing a cache of size C, which can be divided into X different fractions. For instance, if $X = 20$, we can give several fractions of 5% of the cache to each application.

Let p_i be the number of cores on which application A_i is executed, and let x_i be the number of fractions of cache assigned to A_i, for $1 \leq i \leq m$. Hence, A_i uses a cache of size $\frac{x_i}{X}C$. We must have $\sum_{i=1}^{m} p_i = P$ and $\sum_{i=1}^{m} x_i = X$, i.e., all the cores and the cache fractions are partitioned across the applications.

Given p_i and x_i, an application A_i executes one iteration in time $T_i^{real}(p_i, x_i)$. On a given platform, all these values can be measured, and we aim at providing a model that characterizes these values. In the model, we use the following formula:

$$T_i(p_i, x_i) = t_i(p_i)\left(1 + h_i(x_i)\right), \tag{3.1}$$

where $t_i(p_i)$ represents the computation cost and $h_i(x_i)$ the slowdown induced by cache misses in the LLC. Intuitively, the computation cost decreases when p_i increases, and similarly, the slowdown decreases when x_i increases, i.e., $t_i(p_i)$ and $h_i(x_i)$ are non-increasing functions. In this formula, we assume that the slowdown incurred by cache misses does not depend on the number of cores assigned to the application. While this assumption may not be true in practice, we will discuss the model accuracy in Section 3.4, where we measure cache misses and refine the model.

We now detail the model for $t_i(p_i)$ and $h_i(x_i)$.

3.1.1 Computations $t_i(p_i)$

We assume that all applications obey Amdahl's law [3]: $t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}$, where T_i^{seq} is the sequential time of the application executed with 100% of the cache, and s_i is the sequential fraction of the application.

3.1.2 Cache misses effect $h_i(x_i)$

The most challenging part is to model the slowdown factor $h_i(x_i)$. In chip multiprocessors (CMP), many studies have observed that cache miss ratio follows the Power Law, also called the $\sqrt{2}$ rule [54,

68, 101]. The Power Law of cache misses states that for a cache of size C_{act}, the cache miss ratio r can be expressed as

$$r = r_0 \left(\frac{C_0}{C_{act}} \right)^{\alpha}, \tag{3.2}$$

where r_0 represents the cache miss ratio for a baseline cache of size C_0, and α is a parameter ranging from 0.3 to 0.7, with an average at 0.5. We consider $\alpha = 0.5$ in the following.

We slightly generalize the Power Law formula (with $\alpha = 0.5$) to avoid side effects, and define the slowdown as follows:

$$h_i(x_i) = a_i + \frac{b_i}{\sqrt{x_i}}, \tag{3.3}$$

where a_i and b_i are constants depending on the application A_i. From Equation 3.2 with $\alpha = 0.5$, we have $b_i = r_0\sqrt{\frac{C_0 X}{C}}$ (since $C_{act} = \frac{x_i}{X}C$), and a_i is a constant added to avoid side effects. In Section 3.4, we determine a_i and b_i by interpolation, from experimentally measured cache misses, see Table II.

Overall, when assigning p_i cores and a fraction x_i of the cache, and letting $c_i = 1 + a_i$, an application A_i executes one iteration in time:

$$T_i(p_i, x_i) = t_i(p_i) \left(c_i + \frac{b_i}{\sqrt{x_i}} \right). \tag{3.4}$$

3.1.3 Optimization problem

As stated in the introduction of this chapter, the goal is to maximize a weighted throughput, since analysis applications may be required at different rates, from every simulation step to every tenth (or more) step [77]. We let β_i denote the weight of application A_i for $1 \leq i \leq m$. Intuitively, β_i represents the number of times that we should execute application A_i at each iteration step. These priority values are not absolute but relative: for $m = 2$ applications, having $\beta_1 = \frac{1}{4}$ and $\beta_2 = 1$ means we execute four times A_2 (at each step) while executing A_1 only once (every fourth step). This is equivalent to having $\beta_1 = 1$ and $\beta_2 = 4$ if we change the granularity of the simulation steps. In fact, what matters is the relative number of executions of each A_i that is required, hence we aim at maximizing the weighted throughput. The throughput achieved when executing β_i instances of application A_i is $\frac{1}{\beta_i T_i(p_i,x_i)}$, and the objective is to partition the shared cache and assign cores such that the total time taken by the slowest application is minimal, i.e., the lowest weighted throughput is maximal. The weighted throughput allows us to ensure some fairness between applications, and to enforce a better analysis rate of the simulation results whenever the bottleneck is the slowest application. Note that letting $\beta_i = 1$ leads to maximizing

the rate of the analysis when all applications are needed at the same frequency. The optimization problem is formally expressed below:

Definition 3.1 (CoSched-CachePart)**.** *Given m iterative applications with priorities $(A_1, \beta_1), \ldots, (A_m, \beta_m)$ and a platform with P identical cores sharing a memory of size C with X fractions of cache, the* CoSched-CachePart *problem consists in finding a schedule $\{(p_1, x_1), \ldots, (p_m, x_m)\}$ such that*

$$\text{MAXIMIZE} \min_{1 \leq i \leq m} \left\{ \frac{1}{\beta_i T_i(p_i, x_i)} \right\}$$

$$\text{SUBJECT TO} \begin{cases} \sum_{i=1}^{m} p_i = P, \\ \sum_{i=1}^{m} x_i = X. \end{cases}$$

3.2 Scheduling strategies

In this section, we introduce several co-scheduling strategies that we will compare via experiments on the Intel Xeon. We start with a (theoretically) optimal schedule, and then present simple heuristics that we use for comparison.

3.2.1 Optimal solution to CoSched-CachePart

Given the time to execute one iteration of application A_i with p_i cores and a fraction x_i of the cache $T_i(p_i, x_i)$, we can solve the CoSched-CachePart problem optimally, with a dynamic programming algorithm.

Theorem 3.1. CoSched-CachePart *can be solved in time $O(mPX)$, where m is the number of applications, P is the number of processors, and X is the number of different possible cache fractions.*

Proof. Let $T(i, q, c)$ be the maximum weighted throughput that can be obtained with applications $A_1, \ldots, A_i$, using q cores and c fractions of cache. The goal is to find $T(m, P, X)$. We compute $T(i, q, c)$ as follows:

$$T(i,q,c) = \begin{cases} \max\limits_{\substack{1 \leq q_1 \leq q \\ 1 \leq c_1 \leq c}} \frac{1}{\beta_1 T_1(q_1, c_1)} & \text{if } i = 1, \\ \max\limits_{\substack{1 \leq q_i < q \\ 1 \leq c_i < c}} \left\{ \min \left\{ T(i-1, q-q_i, c-c_i), \right. \right. & \\ \qquad \left. \left. \frac{1}{\beta_i T_i(q_i, c_i)} \right\} \right\} & \text{otherwise.} \end{cases}$$

The base case $i = 1$, for one application, takes the best out of all possible allocations (in terms of number of processors and number of cache fractions). Note that for most execution time profile, the execution time in this case is obtained by $T(1, q, c) = \frac{1}{\beta_1 T_1(q,c)}$, since using less processors or less fractions of cache would only increase the execution time, but we write the general expression to encompass any execution time profile, and not only the one given by Equation 3.4.

In the recurrence, we try all possible number of processors and number of cache fractions for application i, and re-use the optimal solution for the $i - 1$ other applications. If we did not use the optimal

solution, we would be able to create a better solution, hence it is easy to see that the problem has an optimal substructure property and can be solved with a dynamic programming algorithm.

There are mPX values to compute, and they can each be obtained in constant time, except for the generalized base case, where we need to perform a maximum over PX values. Overall, with the execution profile of our model, we can compute all values in time $O(mPX)$, and the complexity becomes $O(mP^2X^2)$ in the general case. In practice on the Intel Xeon, we have $m \leq P = 14$, and $X = 20$, hence the dynamic programming algorithm executes almost instantaneously in all the experiments. □

This optimal algorithm provides us with our first strategy to schedule applications, and it is called DP-CP (Dynamic Programming with Cache Partitioning). Checking the behavior of this strategy in practice will assess the accuracy of the performance model, when using the values of $T_i(p_i, x_i)$ obtained with the model of Section 3.1.

3.2.2 Equal-resource assignment

To evaluate the global efficiency of the optimal solution for DP-CP, we compare it to EQ-CP, a simple strategy that allocates the same number of cores and the same number of cache fractions to each application. The algorithm is the following: we start to give $x_i = \left\lfloor \frac{X}{m} \right\rfloor$ and $p_i = \left\lfloor \frac{P}{m} \right\rfloor$ for all i, then, we give the $P \mod m$ extra cores one by one to the first $P \mod m$ applications, and we give the $X \mod m$ extra cache fractions one by one to the last $X \mod m$ applications. Doing this, we forbid the case where an application receives an extra core plus an extra fraction of cache, thereby avoiding a totally unbalanced equal assignment.

3.2.3 Impact of cache allocation

In order to isolate the impact of cache partitioning on performance, we introduce some variants where only the cache allocation is modified:

- DP-EQUAL uses the number of cores returned by the dynamic programming algorithm, hence the same as for DP-CP, but shares the cache equally across applications, as done by EQ-CP.

- We also consider strategies that do not enforce any cache partitioning, but only decide on the number of cores for each application. DP-NOCP uses the same number of cores as DP-CP, and EQ-NOCP uses an equal-resource assignment as in EQ-CP. However, for these two strategies, all applications share the whole cache, i.e., CAT is disabled.

Algorithm 4: Equal allocation with cache partitioning

```
1 EQ-CP (m, P, X) begin
2     for i = 1 to m do p_i ← ⌊P/m⌋; x_i ← ⌊X/m⌋;
3     for i = 1 to P mod m do p_i ← p_i + 1 ;
4     for i = 1 to X mod m do x_{m+1−i} ← x_{m+1−i} + 1 ;
5 end
```

3.3 Experimental setup

In this section, we first describe the platform and the benchmark applications in Section 3.3.1. Then in Section 3.3.2, we explain in details the *Cache Allocation Technology* CAT.

3.3.1 Platform and applications

The experimental platform is composed of a Dell PowerEdge R730 server with two Intel Xeon E5-2650L v4 processors (*Broadwell* microarchitecture). Each processor contains $P = 14$ cores (with Hyper-Threading disabled) sharing a 35MB last-level cache (*Cluster-on-Die* disabled), divided into $X = 20$ slices (or fractions). Nodes run a vanilla 4.11.0 Linux kernel with cache partitioning enabled.

Only one processor (with 14 cores) is used for the experiments, since the LLC is not shared across processors. It matches standard practice because users who co-schedule real-applications often place each application inside a single processor to benefit from the shared cache. Batch schedulers also allocate cores of the same processor whenever possible. Hence our work focuses on co-scheduling the subset of applications that are assigned to a single processor by the user or by the batch scheduler.

Cache experiments are very sensitive to perturbations, so we take great care to ensure that all experiments are fully reproducible. To avoid perturbations: (i) we average values obtained (like cache misses) over 20 (in Section 3.4) or 5 (in Section 3.5) identical runs; (ii) we flush the last-level cache entirely between runs; and (iii) experiments run on a dedicated processor while the program launching and monitoring them runs on the other processor. All the data presented in this chapter (cache misses, number of floating operations, etc), is obtained with PAPI [24].

For validations and performance evaluation, we use six HPC workloads from the NAS benchmarks [11] (see Table I). We consider only NAS benchmarks from class A, as detailed in Table I.

App	Description
CG	Uses conjugate gradients method to solve a large sparse symmetric positive definite system of linear equations
BT	Solves multiple, independent systems of block tridiagonal equations with a predefined block size
LU	Solves regular sparse upper and lower triangular systems
SP	Solves multiple, independent systems of scalar pentadiagonal equations
MG	Performs a multi-grid solve on a sequence of meshes
FT	Performs discrete 3D fast Fourier Transform

Table I: Description of the NAS parallel benchmarks.

3.3.2 Cache Allocation Technology

The Cache Allocation Technology (CAT) [87] is part of a larger set of Intel technologies that are called the Resource Director Technology (RDT) and supported since the *Haswell* architecture. RDT lets the operating system group applications into classes of service (COS). Each class of service describes the amount of resources, in particular cache, that assigned applications can use (see Figure 3.1). Monitoring of current use of these resources may also be available. Currently, resources can be either an amount

of cache or memory bandwidth. In this chapter we will only focus on cache resources (CAT), which implements cache partitioning.

The CAT divides the LLC into X slices of cache. Each COS has a set of slices that applications can use: When reading or writing memory requires to fetch a cache line in the LLC, that cache line must be allocated in the slices available to the class of the current application. However applications may read/modify cache lines that are already available in other slices, for instance when sharing memory between programs in different classes (each cache line can only exist once in the entire cache).

Each slice may only be used by a single class. By default, applications are placed in the default class (COS_0) which contains slices not used by any other class. The set of slices available to a class is a capacity bit-mask (CBM) of length X. With $X = 20$, if COS_1 has access to the last 4 slices (the top 20% of the LLC), CBM_1 would be set to `0xf0000`.

However, CAT has some technical restrictions:

- Number of slices (CBM length) and classes are architecture dependent (20 and 16 on our platform);
- A CBM cannot be empty (each class of applications must have at least one fraction of cache);
- Bits set in a CBM must be contiguous;
- Slices are not distributed geographically in the LLC. Address hashing ensures spreading of slices over the entire LLC. In other words, `0x10000` and `0x00001` CBM should behave exactly the same with respect to locality; there are no NUCA effects (Non Uniform Cache Access).

In this work, we consider a strict cache partitioning, hence each COS contains only one application (and each cache slice is available to a single application).

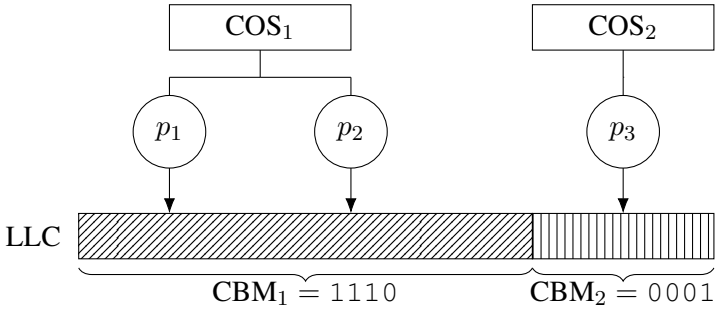


Figure 3.1: CAT example with 2 classes of service, 3 cores and a 4-bit capacity mask (CBM). First COS has 2 cores and 75% of the LLC, the second class of service has the remaining resources.

3.4 Accuracy of the model

In this section, we assess the precision of the model developed in Section 3.1. First, we detail the experimental protocol and explain how to obtain the model parameters for each application in Section 3.4.1. Then, we study in Section 3.4.2 the behavior of cache misses on the platform described in Section 3.3.1, so as to verify whether the Power Law holds for HPC workloads on such architectures. Finally, we study in Section 3.4.3 the accuracy of the model proposed in Section 3.1 by comparing the expected execution time from Equation 3.4 to the measured one.

3.4.1 Experimental protocol

To instantiate the model and check its accuracy, we need to find for each application the value of three parameters used in Equation 3.4: s_i (sequential fraction), a_i (or equivalently $c_i = a_i + 1$), and b_i (cache slowdown). To this purpose, we monitor each application with PAPI [24] and use multiple interpolations on the produced data to find the desired constants. More precisely, we proceed as follows. Each application A_i executes alone on a dedicated processor. First, we give 100% of the cache to the application A_i and vary the number of cores from 1 to 14 to derive the sequential fraction s_i. Then, for each cache fraction x_i ranging from 15% to 85%, we record the number of cache misses when p_i ranges from 1 to 14 and derive values for c_i and b_i. Finally, we put the pieces together, keeping the value of s_i while scaling c_i and b_i by a constant factor, thereby deriving the final values for $T_i(p_i, x_i)$ in Equation 3.4.

As a side note, we point out that this complicated (and definitely not scalable) approach was necessary because the least-square interpolation program would not converge when fed directly with 80% of the 280 experimental values for each application (14 processors, and 16 values of x out of 20). We expect it will be even more challenging to instantiate the model for future platforms where the number of cores will be higher. Note that the Power Law with $\alpha = 0.5$ suits well the behavior of compute-intensive benchmarks such as CG, but struggles to model memory/communication-intensive applications such as MG and FT. The results for each application are displayed in Table II.

App_i	a_i	b_i	s_i
BT	-0.0026	0.0287	0.010
CG	-0.0379	0.0474	0
FT	0.0092	0.0129	0.016
LU	-0.0247	0.0275	0.020
MG	0.0460	0.0073	0.065
SP	-0.0110	0.0254	0.018

Table II: s_i, a_i and b_i obtained by interpolation from the data produced by measurements .

3.4.2 Accuracy of the Power Law

Figure 3.2 shows the evolution of cache miss ratios for the six applications depending on the number of cores and cache fraction. We observe that for most applications, the cache miss ratio increases with the number of cores for small cache fractions, while it does not vary significantly with the number of cores for higher cache fractions. Therefore, these results verify the assumption about the relation between number of cores and cache misses.

On Figure 3.3, we study the evolution of cache miss ratios for each considered application, running alone with a single core. We do not look at cache fractions below $x = 3$ (or 15%) because, according to our experiments, it shows irrelevant results due to cache contention. We observe that the Power Law with $\alpha = 0.5$ suits well the behavior of compute-intensive benchmarks CG, BT, LU and SP, but struggles to model memory/communication-intensive applications like MG and FT.

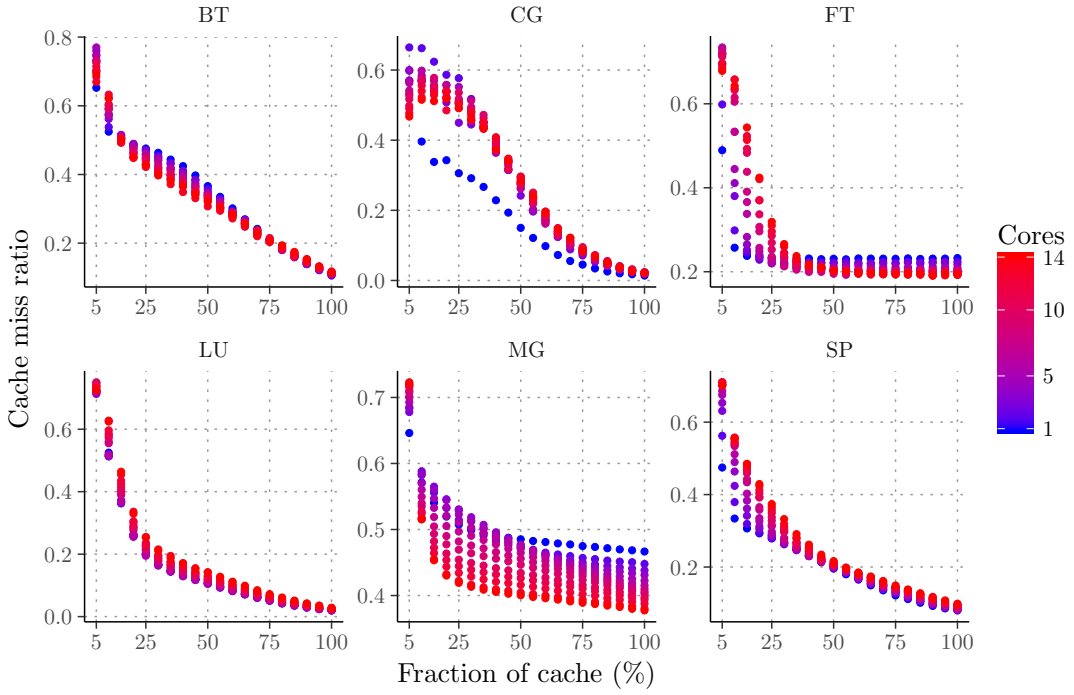


Figure 3.2: Evolution of cache miss ratio when the cache fraction x_i is ranging from 1 to 20 (i.e., from 5% to 100%) and the number of cores p_i is ranging from 1 (blue) to 14 (red).

3.4.3 Accuracy of the execution time

Finally, we aim at verifying the accuracy of the execution time predicted by the model. Figure 3.4 shows, for each application, the comparison between the measured execution time and the model, when the application runs alone on the platform (no co-scheduling here). In Figure 3.4, the number of cores varies from 1 to 14 while the cache fraction is fixed at $x = 3$ (or 15%).

Figure 3.5 shows the relative error between predictions and the real data. The relative error is defined as

$$E_i(p_i, x_i) = \frac{\left|T_i(p_i, x_i) - T_i^{real}(p_i, x_i)\right|}{T_i^{real}(p_i, x_i)},$$

where $T_i^{real}(p_i, x_i)$ is the measured execution time on the cache partitioned platform for application A_i with p_i cores and x_i fractions of cache. We observe that our model predicts execution times rather well for CG and MG, with less than 25% of error for worst cases. For FT, the model is accurate for $x_i \geq 6$ (30%) and $p_i \leq 10$, with a relative error below 15%, but the model loses accuracy for small cache fractions and high number of cores. This is due to a specific behavior of FT: its execution time tends to become constant after a certain core threshold (see Figure 3.4), while the model expects a strictly decreasing execution time. This constant plateau is not due to Amdahl's law (FT is parallel enough to scale up to 14 cores), hence a contention effect (either from the cache or the memory bandwidth) is probably behind this constant level in performance. Another reason to explain these mis-predictions when the number of cores increases, is that the model assumes that the number of cores does not impact LLC cache misses, which is not always true in practice, as seen in Figure 3.2.

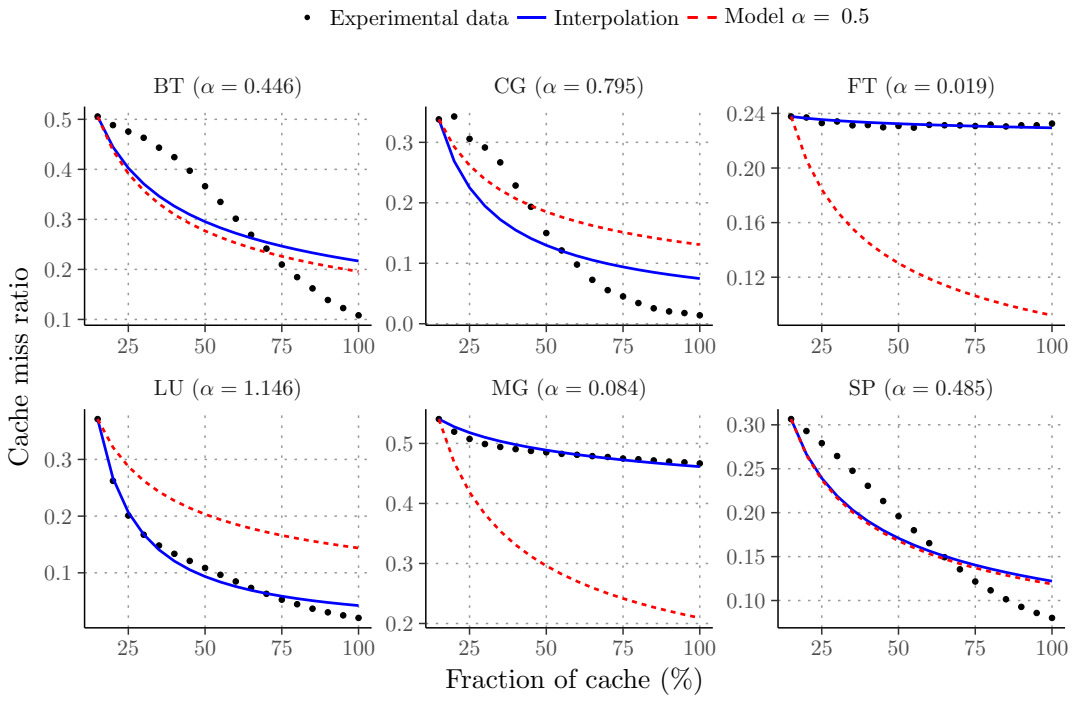


Figure 3.3: Comparison between the predicted cache miss ratio given by the Power Law with $\alpha = 0.5$ in red, the best found α parameter in blue and the measured cache miss ratio in black. Applications run alone on the platform with 1 core.

3.5 Results

To assess the performance of the scheduling strategies of Section 3.2 and to evaluate the impact of cache partitioning on co-scheduling performance, we conduct an extensive campaign of experiments using a real cache partitioned system.

3.5.1 Experimental protocol

The platform and the applications used for all the experiments are described in Section 3.3. Recall that we consider iterative applications, hence we have modified their main loop such that each of them computes for a duration T. We choose a value for T large enough to ensure that each application reaches the steady state with enough iterations (for instance, $T = 3$ minutes for small applications like CG, FT, MG and $T = 10$ minutes for the others). If a co-schedule contains both small and big applications, we use $T = 10$ minutes for all applications. In addition, for all the following experiments, we use 12 cores out of the 14 available, to avoid rounding effects when we co-schedule a number of applications that is not divisible by the number of cores. Similar results were obtained when co-scheduling applications on all 14 cores, in particular with two applications that could use seven cores each.

Evaluation framework. To study the performance of the different algorithms in terms of weighted throughput, we measure the time for one iteration of A_i: $T_i = \frac{T}{\#\texttt{iter}_i}$, where $\#\texttt{iter}_i$ is the number of iterations of application A_i during T. Then, we compute $\min_i \frac{1}{\beta_i T_i}$. We are then interested by the relative speed of each application with respect to the others. Indeed, recall that for all i, j, the goal is

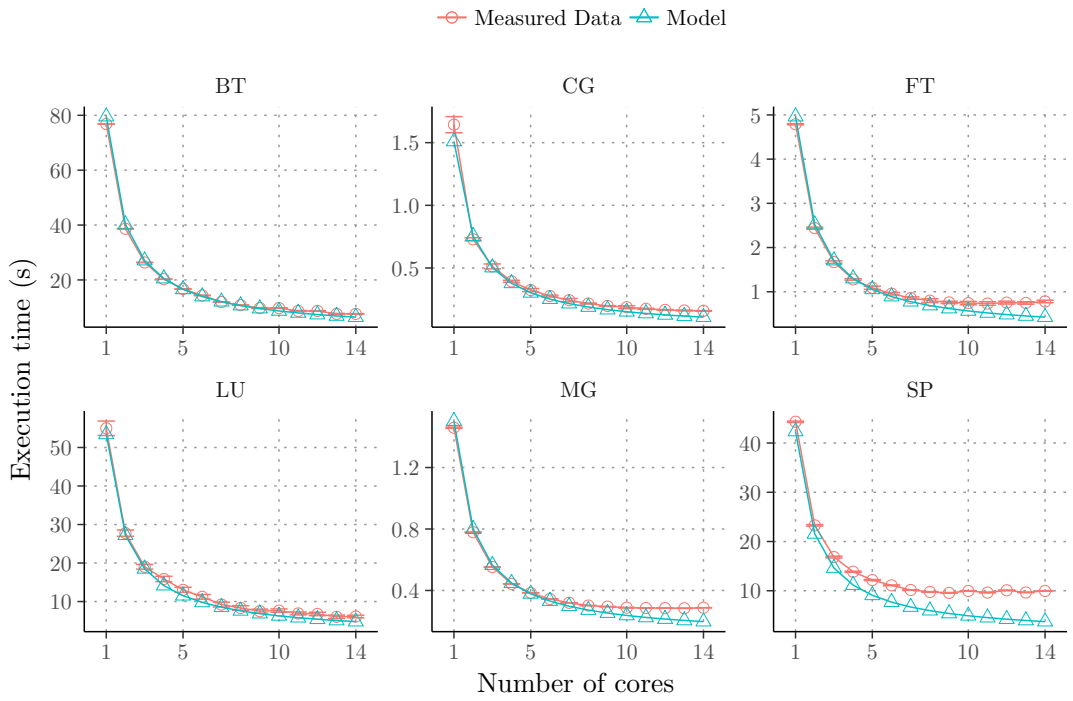


Figure 3.4: Comparison between predicted execution time by the model and measured execution time, when varying the number of cores up to 14 and with a cache fraction set to 15%.

to have $\beta_i T_i = \beta_j T_j$, by definition of the β's. Hence, we further study the following fairness criterion, representing the distance to the optimal fairness, $\Delta_{fairness}$:

$$\Delta_{fairness} = \sum_{i \neq j} \left| \frac{\beta_i T_i}{\beta_j T_j} - 1 \right|. \tag{3.5}$$

In addition to studying the maximum weighted throughput that can be obtained with the applications, we also report the value of $\Delta_{fairness}$ in the experiments, so as to assess whether the heuristics are ensuring that the correct number of iterations of each application is performed during a given amount of time. The goal is to have $\Delta_{fairness}$ as close to 0 as possible.

3.5.2 Impact of cache partitioning

The first step is to assess the impact of cache partitioning (CP) on performance. To this purpose, we co-schedule two applications, so we have three combinations (CG+MG, CG+FT, FT+MG). For all i, j, we set the number of cores for A_i and A_j to six, and we vary the fraction of cache allocated to A_i from 5% to 95% while, at the same time, the cache fraction of A_j is varying from 95% to 5%. The y-axis represents the aggregated number of iterations executed by all applications. We run the applications both with CP enabled, and CP not enabled. Figure 3.6 shows the impact of CP for CG+MG: we can see that when CG has more than 35% of the cache, CP outperforms the version without CP. The impact of CP lies in the behavior of each application, more specifically their data access pattern. CG is a compute intensive application with an irregular memory access pattern, while MG is a memory intensive application. More specifically, MG does not take a great benefit for more cache after 35%, while the performance of CG

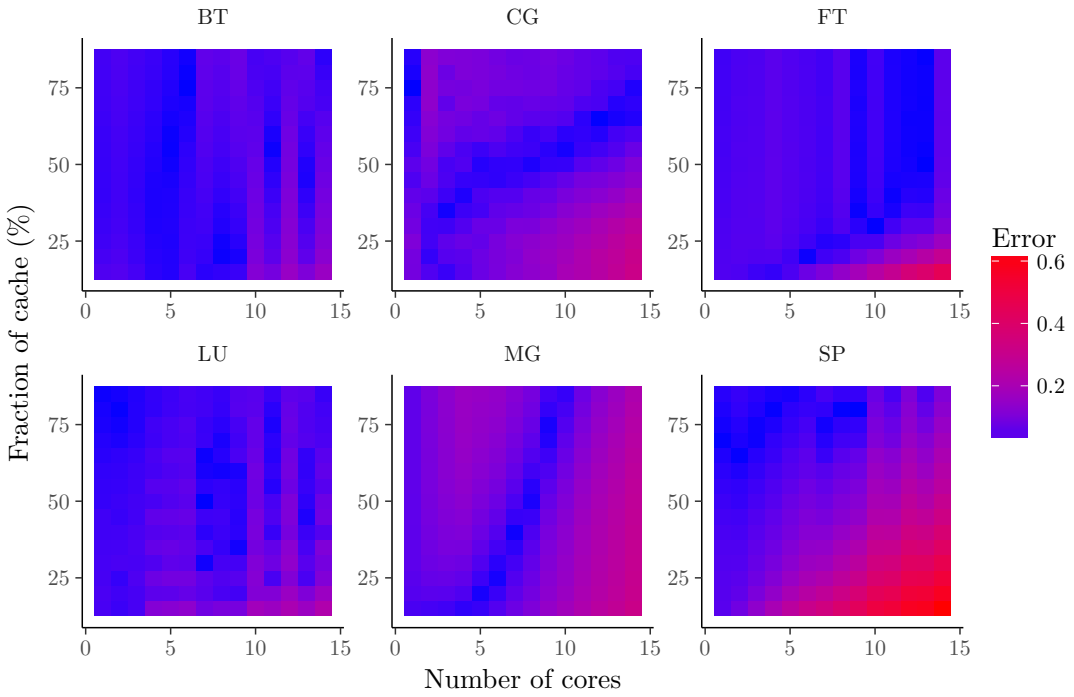


Figure 3.5: Heat-map of the relative error between the model predictions and the measured execution times when the cache fraction is varying from 15% to 85% and the number of cores from 1 to 14.

greatly depends on the cache size (for more details on application behaviors, see Figure 3.2). Without a cache partitioning scheme, by reading/writing a lot of different cache lines, MG will often evict CG cache lines, resulting into a performance degradation of both applications.

Figure 3.7b shows the impact of CP for CG+FT. In this case, we note a small improvement when CG has 80% of the cache. The reason behind this improvement is that FT is more communication intensive (all-to-all communication) than strictly memory intensive, hence the gain obtained by CP is less important than for CG+MG. Since we consider only one processor, the applications that run are the shared memory version (OpenMP), and in that context, the impact of cache on communications is small.

Finally, Figure 3.7a presents the result for the last combination FT+MG. The cache partitioning is not efficient for that combination of two memory and communication intensive applications. If FT has 25% and MG has 75%, then CP can almost achieve the same performance as without CP. This inefficiency is mostly due to the memory intensive and communication intensive behaviors of both applications involved, none of them needs a strict cache partitioning, since their use of the cache varies during iterations.

Summary. The cache partitioning is very interesting when compute-intensive and memory-intensive application are co-scheduled (important gain, up to 25%, for CG+MG, small gain for CG+FT). On the contrary, FT and MG together perform badly with the cache partitioning enabled, these applications do not benefit from the cache to improve their execution time by iteration. Hence, the behavior of applications has a strong impact on the global performance of cache partitioning, and in general, co-scheduling applications with the same behavior results in degraded global performance when using CP.

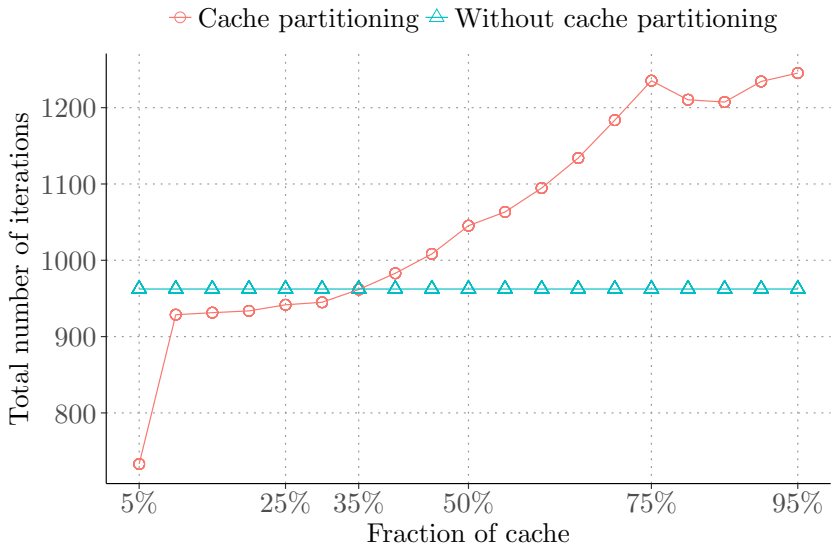


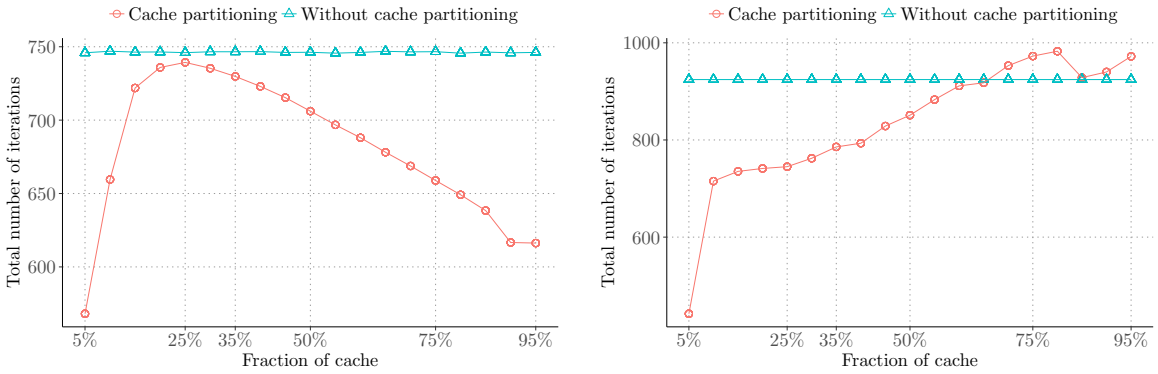
Figure 3.6: CG and MG with 6 cores each, CG has 5% of the cache while MG has the remaining 95%, then CG has 10% and MG 90% and so forth.

3.5.3 Co-scheduling results with two applications

Now that we have demonstrated the interest of cache partitioning, we study the performance of the scheduling strategies of Section 3.2. Recall that the CoSched-CachePart optimization problem aims at maximizing the minimum weighted throughput among co-scheduled applications. Considering two applications (A_i, A_j), for β_i iterations of A_i, we aim at performing β_j iterations of A_j. To avoid some cache effects that appear when the cache area is too small, we set the minimum cache fraction allocated to each application to three (each application has at least 15% of the cache), while the minimum number of cores per application is set to one. We use three different ways to present the result for each studied combination: (i) the objective we want to maximize (minimum weighted throughput), (ii) the ratio of iterations done, and (iii) the $\Delta_{fairness}$ defined in Equation 3.5.

CG+MG. On Figure 3.8a, we see what is the minimum throughput achieved by each method for CG+MG. The weight β associated to MG varies from 0.25 to 4. The algorithms based on dynamic programming DP-CP, DP-Equal and DP-NoCP outperform both equal-resource assignment heuristics Eq-CP and Eq-NoCP. In this scenario, the cache partitioning provides a good performance improvement, since on average DP-CP outperforms DP-NoCP. On the same figure, we also depict the model prediction, which reports the (analytical) minimum throughput computed from $T_i(p_i, x_i)$ values with p_i and x_i derived from the optimal algorithm DP-CP. We observe that the model is accurate enough to satisfactorily fit the performance of DP-CP obtained on the experimental platform.

Figure 3.8b shows the ratio of iterations for CG+MG. Ideally, we would like to obtain $\beta_{\text{CG}}T_{\text{CG}} = \beta_{\text{MG}}T_{\text{MG}}$, the solid black line represents that optimal iteration ratio. First, note that Eq-CP and Eq-NoCP show constant results because they do not depend on weight, but Eq-CP performs better (even without a clever algorithm, cache partitioning helps). Second, we observe that DP-CP is the closest (on average) to the ideal line, hence the cache partitioning really helps here.



(a) FT and MG

(b) CG and FT

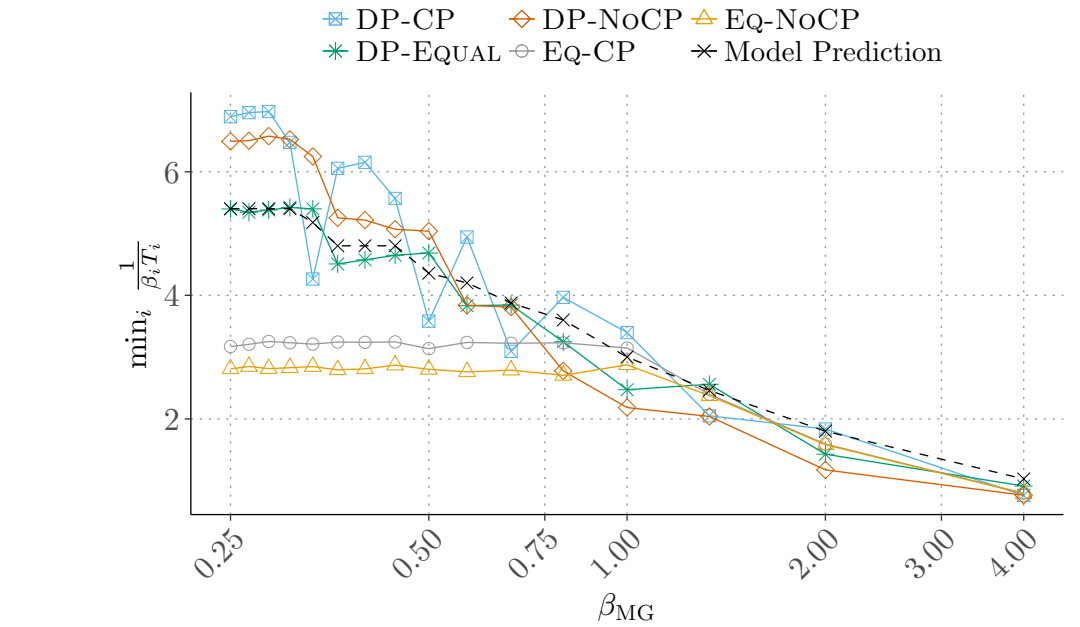
Figure 3.7: CG co-scheduled with MG or FT, with 6 cores each.

Finally, Figure 3.8c presents the $\Delta_{fairness}$, as defined in Equation 3.5. We observe that DP-CP, DP-NoCP and DP-Equal exhibit the same $\Delta_{fairness}$, near to zero, while Eq-CP and Eq-NoCP are far from the optimal fairness.

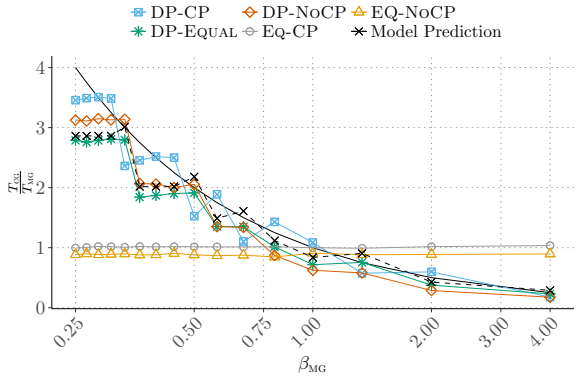
CG+FT. In Figure 3.9a, we observe that DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP when β_{FT} is larger than 0.5. Only, DP-NoCP outperforms Eq-NoCP all the time. When β_{FT} is smaller than 0.5, the two variants without cache partitioning perform better than the two versions with cache partitioning. As explained in Section 3.5.2, due to its communication-intensive behavior, FT will not benefit a lot from cache partitioning techniques. Figure 3.9b presents the iteration ratio (i.e., the fairness among co-scheduled applications) when we co-schedule CG+FT: DP-CP, DP-Equal and DP-NoCP exhibit good performance, and we are very close to the black line that represents the ideal iteration ratio to reach. On Figure 3.9c, we observe the fairness criterion: Eq-CP and Eq-NoCP show an important $\Delta_{fairness}$ as expected, and DP-CP, DP-Equal and DP-NoCP show the same good performance, very close to zero. As for CG+MG, for this case we notice that the model is close enough of the performance of DP-CP.

MG+FT. Figure 3.10a presents the results obtained for MG+FT. DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP, except for β_{FT} lower than 0.50. For both DP-CP and Eq-CP, the cache partitioning does not bring a important improvement. The main reason is that co-scheduling one memory and one communication intensive application is not very efficient (see Section 3.5.2). Figure 3.10b shows that DP-CP, DP-Equal and DP-NoCP perform well, very close to the ideal iteration ratio (the solid black line). On Figure 3.10c, we note that the $\Delta_{fairness}$ is close to zero for DP-CP, DP-Equal and DP-NoCP, while (logically) the $\Delta_{fairness}$ is larger for Eq-CP and Eq-NoCP.

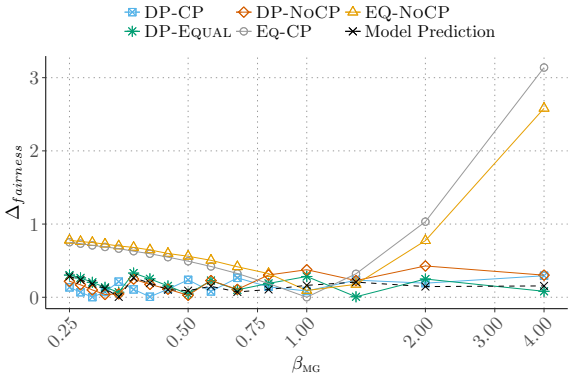
BT, LU, SP co-scheduled with MG. Figures 3.11 to 3.13 show the minimum throughput (on the left) and the error norm (on the right) obtained by co-scheduling, respectively, BT+MG, LU+MG and SP+MG. For the minimal throughput (on the left of each figure), both results are quite similar, all variants based on our algorithm DP-CP outperform Eq-CP and Eq-NoCP. The cache partitioning does not bring a significant gain in this scenario, but DP-CP is always better than DP-NoCP. We observe that DP-Equal perform always worst than DP-CP and DP-NoCP, which means that doing a naive cache partitioning (an equal one in that case) can lead to important performance degradations. For this



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).

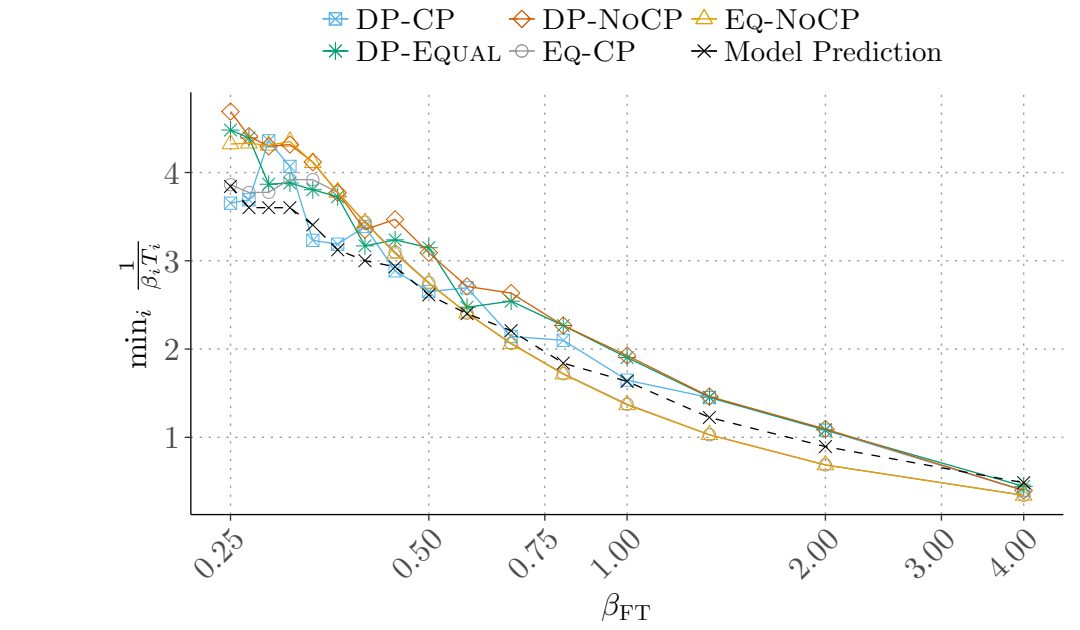


(c) Relative error from the objective (lower is better).

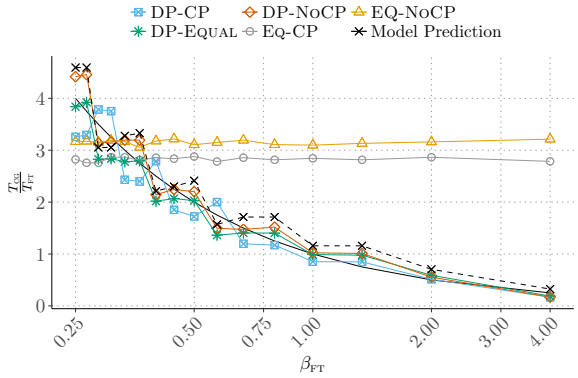
Figure 3.8: CG and MG when β_{MG} is varying from 0.25 to 4.

scenario, because of the high values of the $\Delta_{fairness}$ (respectively 0.25 and 0.4 for the best cases), we only present the fairness criterion $\Delta_{fairness}$. Indeed, BT, LU and SP are much larger than MG in terms of number of operations (by roughly 10^3), hence it is impossible to do, for example, four times more iterations of LU than iterations of MG without a very large value of T.

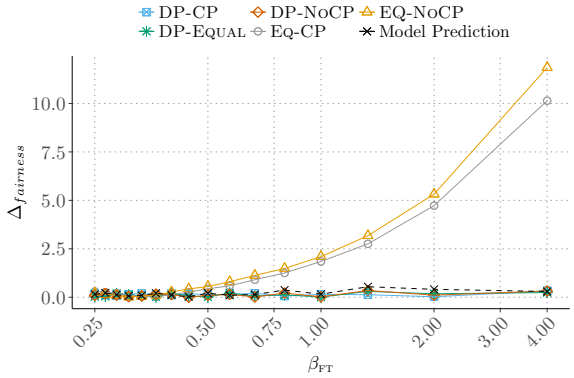
Special case: CG and MG when each application has six cores. We are now interested into a special case: how the cache will affect co-scheduling performance. All applications have the same number of cores (six in our case), so only the cache is available to increase performance. Figure 3.14a shows the global performance of all methods. Obviously, only DP-CP takes advantage of this scenario because only this method can choose how to partition the cache. If β_{MG} is smaller than 1, it means that we have to compute more CG than MG, and in that case, the cache has a strong effect (up to 25% improvement with cache partitioning enabled). We also observe that the model prediction is pretty close



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).

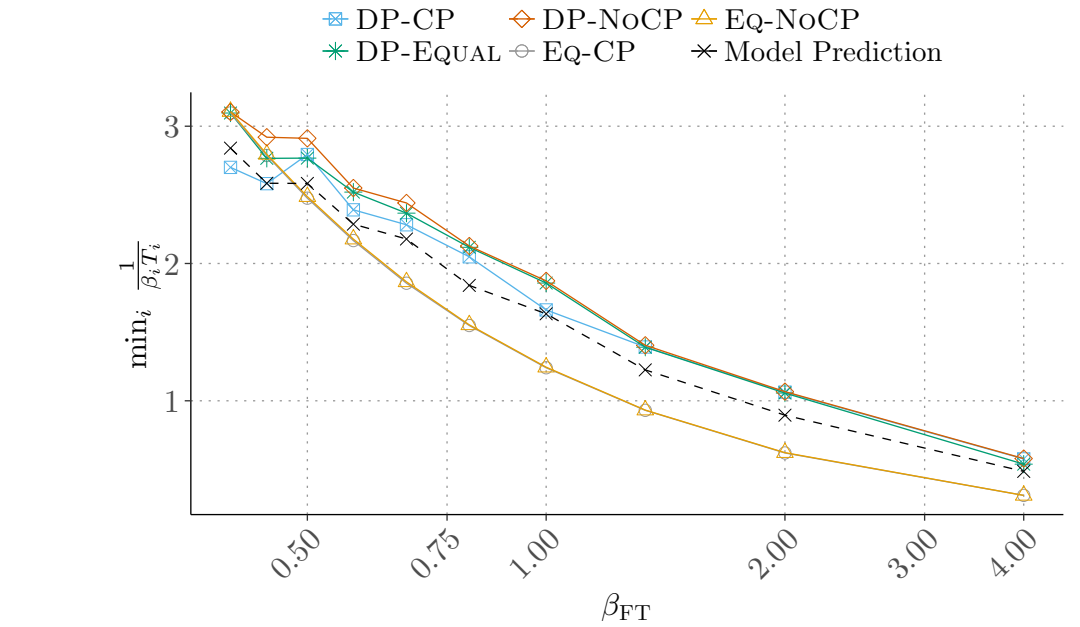


(c) Relative error from the objective (lower is better).

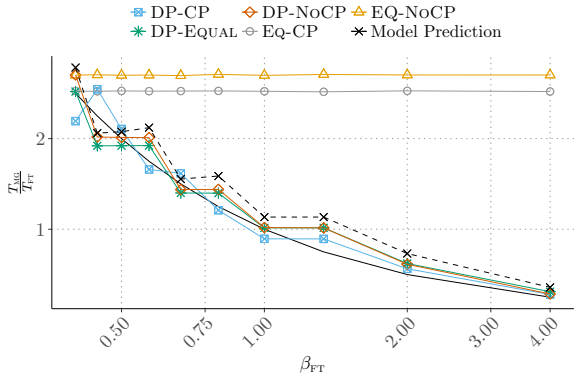
Figure 3.9: CG and FT when β_{FT} is varying from 0.25 to 4.

to the experimental results. With this scenario, we are able to isolate which part of performance relies on cache effect. Figure 3.14b depicts the iteration ratio achieved with an equal number of cores for each application. We observe that with only the cache, it is hard to enforce the required ratio of the number of iterations, according to the values of the β_i. Figure 3.14c represents the fairness criterion$\Delta_{fairness}$ between the ideal iteration ratio and the iteration ratio obtained with each method. Note that the $\Delta_{fairness}$ is high for every method, but the error of DP-CP is the smallest.

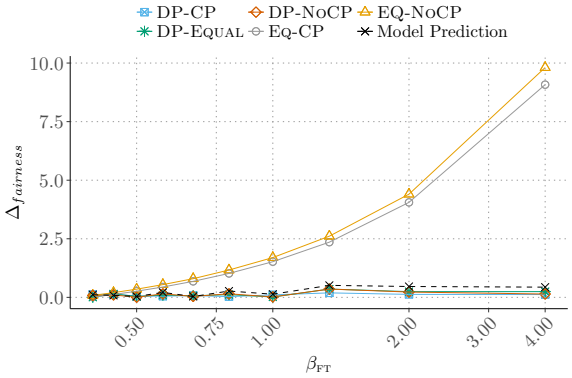
Summary. The model is accurate enough to enforce that the corresponding optimal DP algorithm performs well: in most cases, DP-CP, DP-Equal and DP-NoCP outperform Eq-CP and Eq-NoCP. On the cache partitioning side, when co-scheduling CG and MG, the cache partitioning is really interesting to isolate applications that pollute the cache, such as MG. Figure 3.14a clearly shows the impact of cache on performance when the number of cores is set for each application. In the worst cases, for



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).



(c) Relative error from the objective (lower is better).

Figure 3.10: MG and FT when β_{FT} is varying from 0.25 to 4.

instance with FT and MG, the cache partitioning does not improve performance, but does not degrade it either.

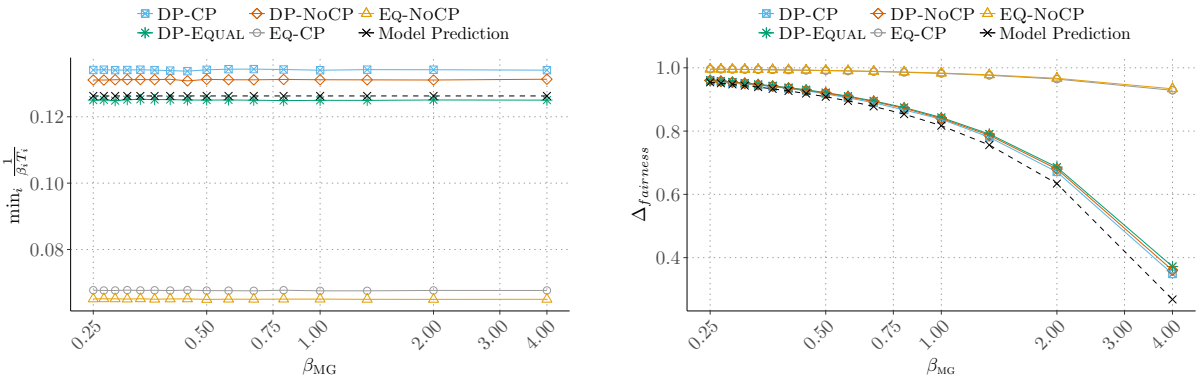


Figure 3.11: Minimum throughput and $\Delta_{fairness}$ for BT+MG.

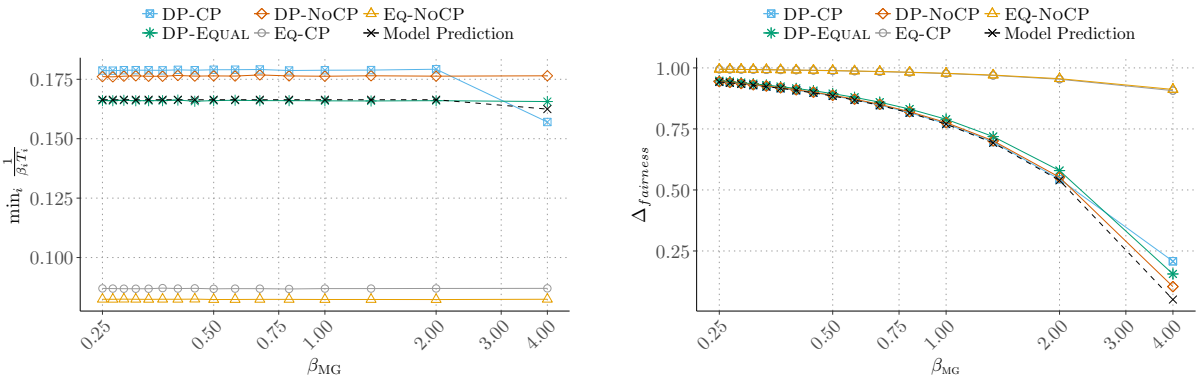


Figure 3.12: Minimum throughput and $\Delta_{fairness}$ for LU+MG.

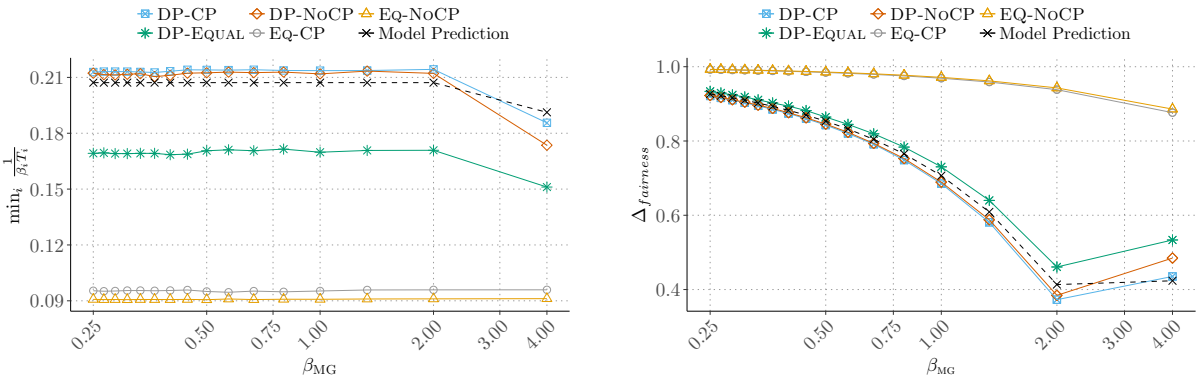
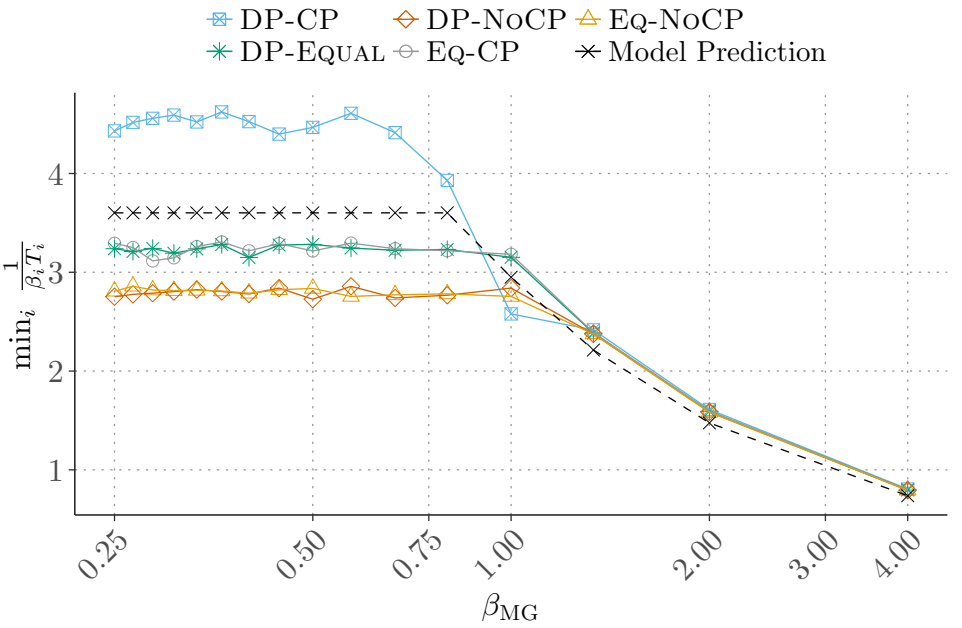
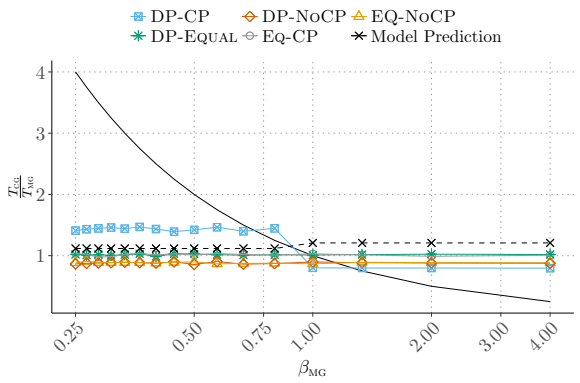


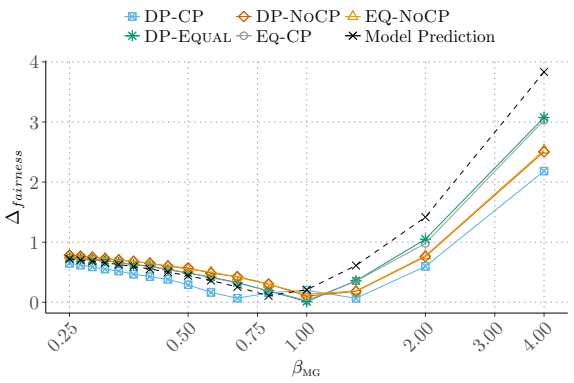
Figure 3.13: Minimum throughput and $\Delta_{fairness}$ for SP+MG.



(a) Minimum throughput (higher is better).



(b) Iteration ratio done (closer to the solid black line is better).



(c) Relative error from the objective (lower is better).

Figure 3.14: CG and MG when β_{MG} is varying from 0.25 to 4 and when both applications have six cores.

3.5.4 Co-scheduling results with three applications

In this section, we present the results with three co-scheduled applications. Similarly to the case with two applications, with three applications (A_1, A_2, A_3), only β_3 is ranging from 0.25 to 4, while $\beta_1 = \beta_2 = 1$. First, we focus only on co-schedules with CG and MG, because they are very interesting applications to study. Second, we study all combinations of co-scheduling with CG, FT and MG. We do not look at the iteration ratio in this section, but focus on minimum throughput and the $\Delta_{fairness}$.

2CG+MG. Figure 3.15 shows the minimum throughput obtained when we co-schedule 2CG+MG, while the weight associated to MG is ranging from 0.25 to 4. Note that it It is interesting to co-schedule multiple copies of the same application (two CGs in this scenario) in order to to improve the global efficiency, when this application exhibits a speedup profile with limited gain from adding extra cores and/or extra fractions of caches. We observe that the scheduling strategies building on the dynamic programming algorithm DP-CP, DP-EQUAL and DP-NOCP outperform EQ-CP and EQ-NOCP. In addition, cache partitioning shows a great interest here: DP-CP exhibits a gain around 15% on average over DP-NOCP and DP-EQUAL. The $\Delta_{fairness}$ is also depicted on the right. Recall that ideally, we would like to have $\beta_i T_i = \beta_j T_j$ for all i, j (see Equation 3.5). We observe that the method that is the closest to zero is DP-CP, confirming the strong interest of cache partitioning.

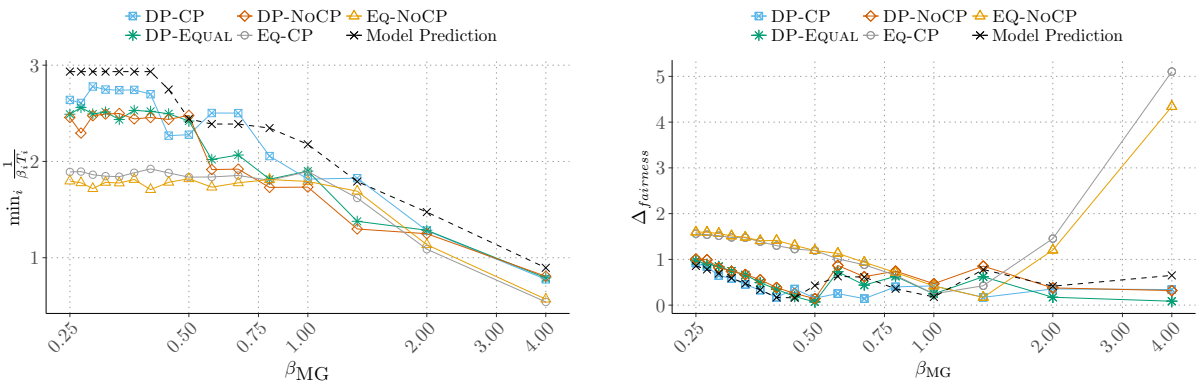


Figure 3.15: Minimum throughput and $\Delta_{fairness}$ for 2CG+MG.

2MG+{CG, BT, LU, SP}. Figure 3.16 presents the minimal throughput obtained by each method when we co-schedule 2MG+CG, where the weight of CG is ranging from 0.25 to 4. Again, the DP-based strategies DP-CP, DP-EQUAL and DP-NOCP exhibit good performance for β_{CG} smaller than 0.50, but they suffer from a lack of performance when β_{CG} is between 0.50 and 1. When β_{CG} is larger than 1, DP-CP becomes the best method again. On the right of Figure 3.16, we can see the confirmation that the proposed dynamic programming algorithm is the method that minimizes the best $\Delta_{fairness}$, even though the cache partitioning with DP-CP and DP-EQUAL does not bring any clear advantage in this scenario. This is mainly due to the fact that the application with the varying weight is a compute-intensive application, co-scheduled with two memory-intensive applications. According to our experiments, when compute-intensive applications are outnumbered by memory-intensive applications, the cache partitioning is often less efficient.

Figures 3.17 to 3.19 also presents, the minimal throughput obtained when we co-schedule, respectively, 2MG+BT, 2MG+LU and 2MG+SP. 2MG co-scheduled with BT, LU or SP lead to the same behavior for the minimum throughput and the $\Delta_{fairness}$, the variants based on our dynamic algorithm

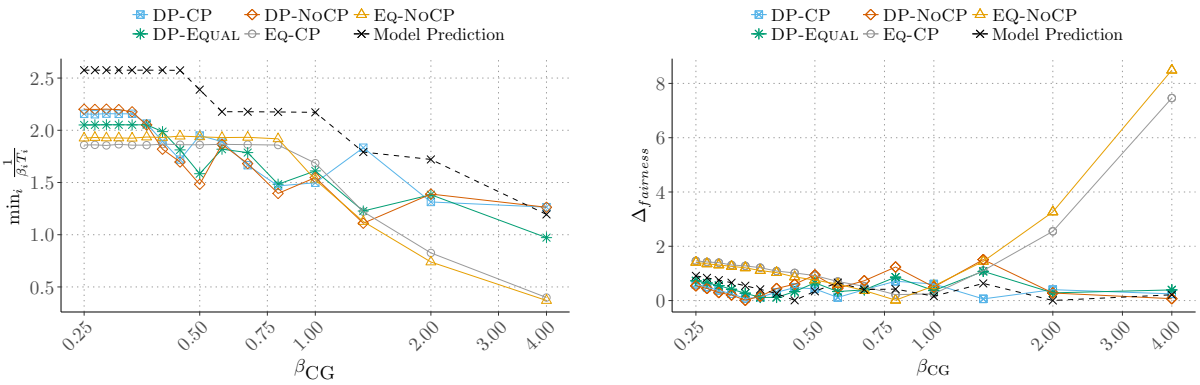


Figure 3.16: Minimum throughput and $\Delta_{fairness}$ for 2MG+CG.

DP-CP, DP-EQUAL and DP-NOCP perform better than EQ-CP and EQ-NOCP. The error norm, for the three cases, is very important. The reason behind the important values of the error norm is that MG is very small compared to LU, BT and SP.

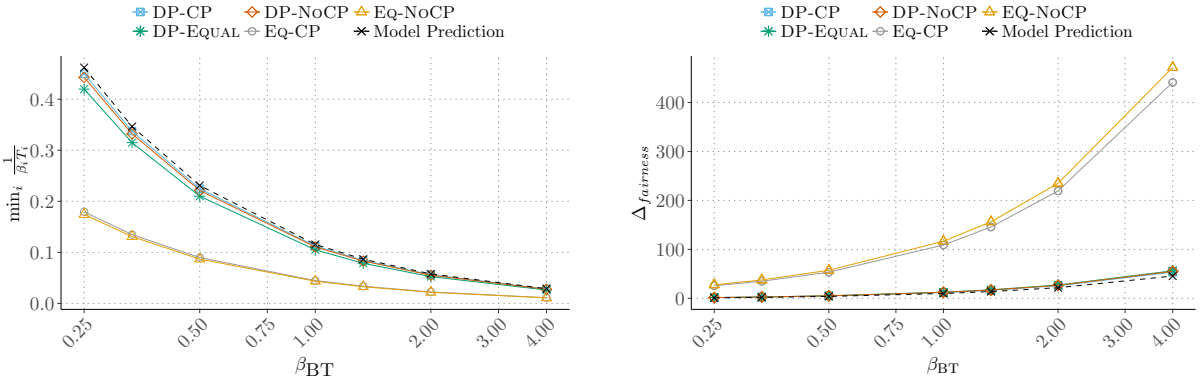


Figure 3.17: Minimum throughput and $\Delta_{fairness}$ for 2MG+BT.

CG+MG+FT. Figure 3.20 shows the minimum throughput obtained when co-scheduling the three different applications, while varying only the weight β_{FT} of FT. We observe that the performance of the three DP-based algorithms is close to the performance obtained with the equal-resource assignment for β_{FT} smaller than 0.5, but for the other cases, DP-CP and all its variants outperform EQ-CP and EQ-NOCP. $\Delta_{fairness}$ leads to the same conclusion: DP-CP, DP-NOCP and DP-EQUAL are much closer to zero than EQ-CP and EQ-NOCP, especially when β_{FT} is larger than 0.5.

Next, Figure 3.21 is the counterpart of Figure 3.20 when varying only the weight β_{MG} of MG. The results obtained by the DP-based algorithms are very good with an average gain around 50% over the EQ-CP variants, especially when β_{MG} is below 1. We note that the cache partitioning does not take advantage of this scenario, DP-CP shows degraded performance compared to DP-NOCP. For the $\Delta_{fairness}$, the method that performs best is DP-CP, close to DP-NOCP and DP-EQUAL though.

Finally, Figure 3.22 is the counterpart of Figure 3.20 and Figure 3.21 when varying only β_{CG}. The behavior of all DP-CP variants is interesting: for $0.25 \leq \beta_{CG} \leq 0.44$, the resource allocation, both for cores and cache, does not change, resulting into the decreasing of the minimum weighted throughput when β_{CG} is increasing (so $\frac{1}{\beta_{CG} T_{CG}}$, which is actually the minimum here, is decreasing). At $\beta_{CG} =$

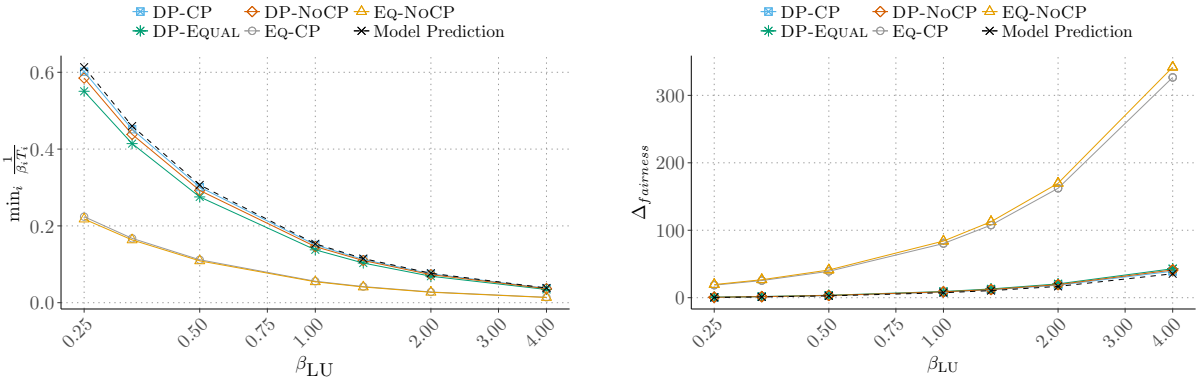


Figure 3.18: Minimum throughput and $\Delta_{fairness}$ for 2MG+LU.

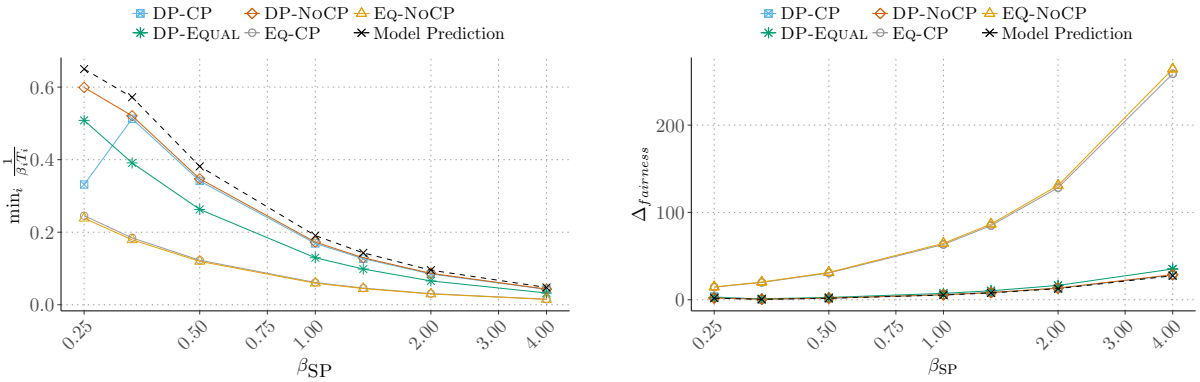


Figure 3.19: Minimum throughput and $\Delta_{fairness}$ for 2MG+SP.

0.5, the allocation of resources changes for DP-CP variants (more and more resources are allocated to CG, in order to fit the increasing requirement). We observe that DP-CP, DP-EQUAL and DP-NOCP logically outperform EQ-CP and EQ-NOCP to maximize the minimum weighted throughput among the co-scheduled applications. However, the cache partitioning does not help in this scenario, mainly because we vary the weight of the only compute-intensive application. In terms of $\Delta_{fairness}$, obviously DP-CP, DP-EQUAL and DP-NOCP perform better than EQ-CP and EQ-NOCP. Among DP-CP, DP-EQUAL and DP-NOCP, we see that the cache partitioning version is the best method to minimize the $\Delta_{fairness}$.

Summary. Overall, we showed that we can obtain important gains using cache partitioning (CP) when co-scheduling three applications, but it is not always the case. The difficulty of obtaining some gain with CP increases with the number of applications involved. The first reason lies in the cache size, often too small to be efficiently partitioned between the applications. The second reason is related to the behavior of the co-scheduled applications. The results show that co-scheduling one or two compute-intensive applications, such as CG, plus one memory-intensive application, such as MG, is a good way to achieve significant improvements with CP. CG is a compute-intensive kernel that performs a lot of irregular memory accesses, while MG is a memory-intensive kernel, hence if we co-schedule one CG and one MG, MG will evict very often cache lines belonging to CG, which will slow down its execution.

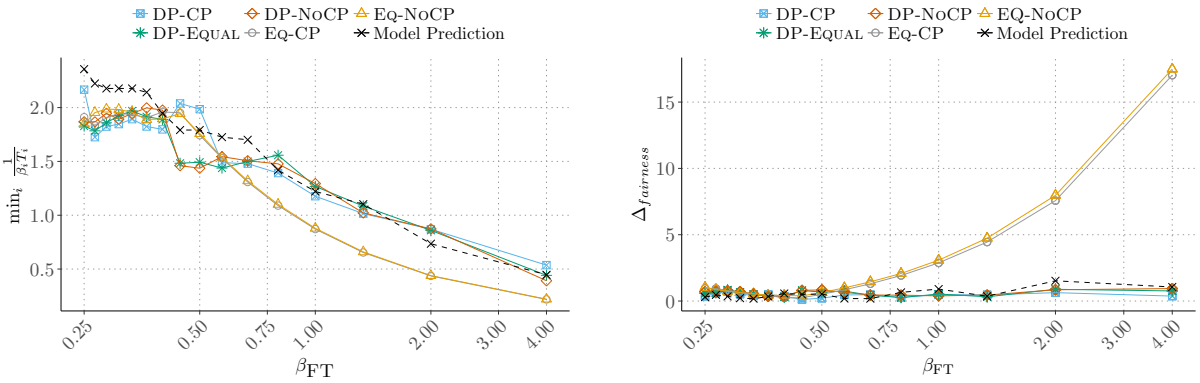


Figure 3.20: Minimum throughput and $\Delta_{fairness}$ for CG, MG and FT.

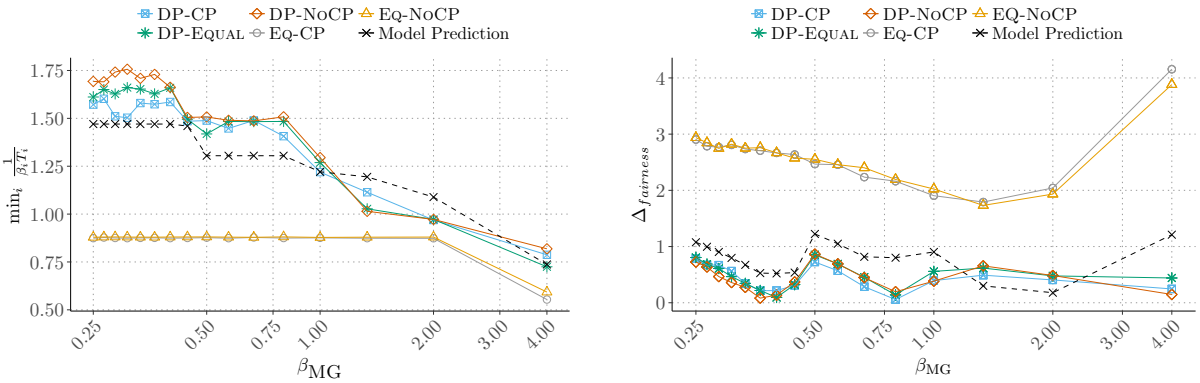


Figure 3.21: Minimum throughput and $\Delta_{fairness}$ for CG, FT and MG.

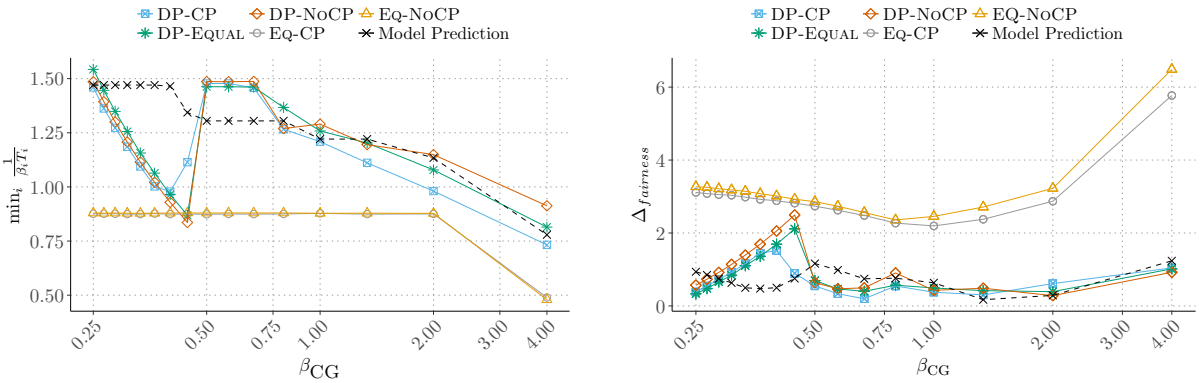


Figure 3.22: Minimum throughput and $\Delta_{fairness}$ for MG, FT and CG.

3.6 Conclusion

We have investigated the problem of co-scheduling iterative HPC applications, using the CAT technology provided by Intel to partition the cache. We have proposed a model for the execution time of each application, given a number of cores and a fraction of cache, and we have shown how to instantiate the model on applications coming from the NAS benchmarks. The model turns out to be accurate, as shown in the experiments where we compare the execution time predicted by the model to the real execution time. Several scheduling strategies have been designed, with the goal to maximize the minimum weighted throughput of each application. In particular, we have introduced an optimal strategy for the model, based upon a dynamic programming algorithm. The results demonstrate that in practice, the optimal strategy often leads to better results than a naive strategy sharing equally the resources between applications. Also, we have determined which combinations of applications benefit most from cache partitioning, and demonstrated the usefulness of cache partitioning.

Future work will be devoted to extending this experimental study. We hope to get access to platforms with larger shared caches, so that we could scale up the experiments and confirm the usefulness of cache partitioning techniques. The first research direction is to design a better interpolation strategy, capable of retro-fitting a subset of the experimental data (execution times for each application, with each processor number and cache fraction) into a simple formula like Equation 3.4, and with good precision. We will also generalize the experiments to multiprocessors and see if there is a benefit in moving applications from one processor to another, in order to avoid co-locating several cache-intensive applications on the same processor. Another interesting direction would be to consider the Universal Scalability Law [52] instead of Amdahl's law, thereby generalizing the model in order to account for contentions.

Chapter 4

Resilient co-scheduling of malleable applications

In Chapters 2 and 3, we have been focusing on co-scheduling with memory aspects (last-level cache). In this chapter, we focus on another challenge that must be addressed at scale: resilience. To the best of our knowledge, co-scheduling has been investigated so far only in the context of fault-free platforms. However, large-scale platforms are prone to failures. Indeed, for a platform with p processors, even if each node has an individual MTBF (Mean Time Between Failures) of 120 years, we expect a failure to strike every $120/p$ years, for instance every hour for a platform with $p = 10^6$ nodes. Failures are likely to destroy the load-balancing achieved by co-scheduling algorithms: if all applications were assigned resources by the co-scheduler so as to complete their execution approximately at the same time, the occurrence of a failure will significantly delay the completion time of the corresponding application. In turn, several failures may well create severe imbalance among the applications, thereby significantly degrading performance.

To cope with failures, the de-facto general-purpose error recovery technique in HPC is checkpoint and rollback recovery [43]. The idea consists in periodically saving the state of the application, so that when an error strikes, the application can be restored into one of its former states. The most widely used protocol is coordinated checkpointing, where all processes periodically stop computing and synchronize to write critical application data onto stable storage. The frequency at which checkpoints are taken should be carefully tuned, so that the overhead in a fault-free execution is not too important, but also so that the price to pay in case of failure remains reasonable. Young and Daly provide good approximations of the optimal checkpointing interval [33, 122].

This chapter investigates co-scheduling on failure-prone platforms. Checkpointing helps to mitigate the impact of a failure on a given application, but it must be complemented by redistributions to re-balance the load among applications. Co-scheduling usually involves partitioning the applications into *packs*, and then scheduling each pack in sequence, as efficiently as possible. We focus on co-scheduling a given pack of applications that execute in parallel, and leave the partitioning for further work. This is because scheduling a given pack becomes a difficult endeavor with failures (and redistributions), while it was of linear complexity without failures. Also, designing efficient pack scheduling algorithms is needed whenever there are relatively few applications that can be all scheduled simultaneously, and it is a prerequisite before tackling the general problem. Given a pack, i.e., a set of parallel applications that start execution simultaneously, there are two main opportunities for redistributing processors. First, when an application completes, the applications that are still running can claim its processors. Second, when a failure strikes an application, that application is delayed. By adding more resources to it, we can reduce its final completion time. However, we have to be careful, because each redistribution has a cost,

which depends on the volume of data that is exchanged, and on the number of processors involved in redistribution. In addition, adding processors to an application increases its probability to fail, so there is a trade-off to achieve in order to minimize the expected completion time of the pack.

Main contributions. In this chapter, we provide the design of a detailed and comprehensive model for scheduling a pack of applications on a failure-prone platform. We prove that the problem is NP-complete for malleable applications, even in a fault-free context. Therefore, we design polynomial-time heuristics that perform redistributions and account for processor failures. A fault simulator is used to perform extensive simulations that demonstrate the usefulness of redistribution and the performance of the proposed heuristics.

The rest of the chapter is organized as follows. First, we discuss related work in Section 4.1. The model and the optimization problem are formally defined in Section 4.2. In Section 4.3, we expose the complexity results. We introduce some polynomial-time heuristics in Section 4.4, which are assessed through simulations using a fault generator in Section 4.5. Finally, we conclude and provide directions for future work in Section 4.6.

4.1 Related work

4.1.1 Parallel application models

A parallel application is an application that may use several processors during its execution. Note that the scheduling literature uses the term *parallel tasks* rather than *parallel application*. Many parallel application models have been developed, and several types of applications have been defined. In 1986, with the development of multiprocessor systems, Błażewicz et al.[17] have modeled the problem of scheduling a set of independent parallel applications on identical processors. The number of processors assigned to each application was fixed during the execution. They showed that the problem is NP-complete when the number of processors is not fixed. An application that has a fixed number of processors is called *rigid*. In 1989, Du and Leung [40] have developed a model called the Parallel Task System, where an application is executed by one or more processors at the same time, but the number of processors assigned to one application cannot exceed a certain threshold. Contrarily to the Błażewicz's model, the number of processors is not fixed in advance, but once it is determined (between one and the threshold), it remains fixed during the execution. Such applications are called *moldable*. Finally, a *malleable* application can have its number of allocated processors vary during the execution. Błażewicz et al. [18] have designed approximation algorithms to solve the problem of scheduling independent malleable applications. Malleable applications are more flexible than rigid and moldable applications, and they can be implemented with data redistribution techniques (the technique used in this chapter) or work stealing. In practice, changing the number of processors at runtime requires specific tools, frameworks and even dedicated programming languages like Cilk [46]. Martín et al. [79] have developed an MPI extension, called Flex-MPI, which introduces malleability in MPI. Flex-MPI can achieve a load balancing among applications through a prediction model. The prediction model in Flex-MPI does not take into account resilience aspects.

One contribution of this work is to develop a complete model taking into account resilience aspects. We also provide heuristics able to re-assign processors to applications that need them. We also show that the problem of finding a schedule that minimizes the execution time with fixed redistribution costs and without failures is NP-complete (in the strong sense).

4.1.2 Resilience

One of the most used technique to handle fail-stop errors in HPC is checkpoint and rollback recovery [43]. The idea is to periodically save the system state, or the application memory footprint onto a stable storage. Then, after a downtime and a recovery time, the system can be restored into a former valid state (rollback step). Another technique to dealing with fail-stop errors is process replication, which consists in replicating a process and even replicate communications. For instance, the project RedMPI [45] implements a process replication mechanism and quadruplicates each communication.

In this chapter, we use a light-weight checkpointing protocol called the *double checkpointing algorithm* [36, 88]. This is an in-memory checkpointing protocol, which avoids the high overhead of disk checkpoints. Processors are paired: each processor has an associated processor called its *buddy processor*. When a processor stores its checkpoint file in its own memory, it also sends this file to its buddy, and the buddy does the same. Therefore, each processor stores two checkpoints, its own and that of its buddy. When a failure occurs, the faulty processor looses these two checkpoint files, and the buddy must re-send both checkpoints to the faulty node. If a second failure hits the buddy during this recovery period (which happens with very low probability), we have a fatal failure and the system cannot be recovered.

4.1.3 Co-scheduling algorithms

This chapter provides an important extension to a previous work on co-schedules [9], which already demonstrated that sharing the platform between two or more applications can lead to significant performance and energy savings [105]. To the best of our knowledge, it is the first work to consider co-schedules and failures, and hence to use malleable applications to allow redistributions of processors between applications. However, we point out that co-scheduling with packs can be seen as the static counterpart of batch scheduling techniques, where jobs are dynamically partitioned into batches as they are submitted to the system (see [84] and the references therein). Batch scheduling is a complex online problem, where jobs have release times and deadlines, and where only partial information on the whole workload is known when taking scheduling decisions. On the contrary, co-scheduling applies to a set of applications that are all ready for execution. In this chapter, as already mentioned, we restrict to a single pack, because scheduling already becomes difficult for a single pack with failures and redistributions.

Contrarily to Chapters 2 and 3, in this chapter we do not consider interferences induced by co-scheduled applications. A more detailed survey on co-scheduling techniques can be found in Chapter 2, Section 2.1.

4.2 Framework

We consider a pack of n independent malleable applications $\{T_1, \ldots, T_n\}$, and an execution platform with p identical processors subject to failures. We assume $n \leq 2p$ due to the use of the double checkpointing model. The objective is to minimize the expected completion time of the last application. First, we define the fault model in Section 4.2.1. Then, we show how to compute the execution time of an application in Section 4.2.2, assuming that no redistribution has occurred. The redistribution mechanism and its associated cost are discussed in Section 4.2.3. Finally, the objective function is detailed in Section 4.2.4.

4.2.1 Fault model

We consider fail-stop errors, which are detected instantaneously. To model the rate at which faults occur on one processor, we use an exponential probability law of parameter λ. The mean (or MTBF) of this

law is $\mu = \frac{1}{\lambda}$. The MTBF of an application depends upon the number of processors it is using, hence changes whenever a redistribution occurs. Specifically, if application T_i is (currently) executed on j processors, its MTBF is $\mu_{i,j} = \frac{\mu}{j}$ (see [58, Proposition 1.2] for a proof). To recover from fail-stop errors, we use the double checkpointing scheme, or *buddy algorithm* [36, 88] (see Figure 4.1).

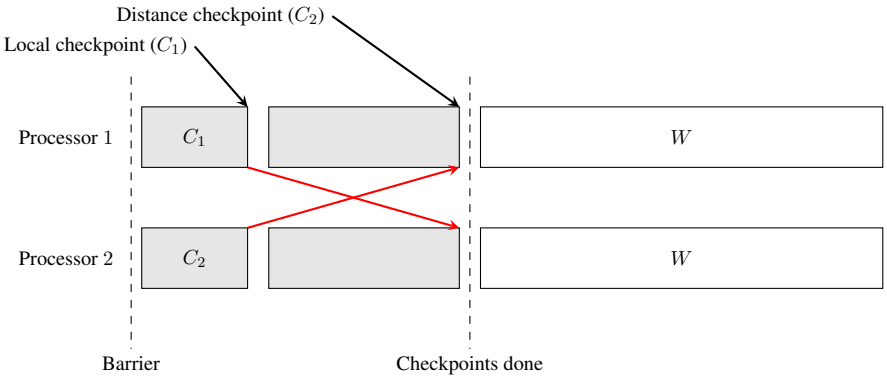


Figure 4.1: Double checkpointing scheme example.

Therefore, the number of processors assigned to each application must be even. We enforce periodic checkpointing for each application. Formally, if application T_i is executed on j processors, there is a checkpoint every period of length $\tau_{i,j}$, with a cost $C_{i,j}$.

We now explain how to compute the cost $C_{i,j}$ of a checkpoint when application T_i executes with j processors. Recall that we use in-memory checkpointing. Let m_i be the memory footprint (total data size) of application T_i. Each of the j processors holds $\frac{m_i}{j}$ data, which it must send to its buddy processor. The time to communicate a message of size s is $\beta + \frac{s}{\tau}$, where β is a start-up latency and τ the link bandwidth. We derive that

$$C_{i,j} = \frac{m_i}{j\tau} + \beta.$$

As for the checkpointing period $\tau_{i,j}$, we use Young's formula [122] and let

$$\tau_{i,j} = \sqrt{2\mu_{i,j}C_{i,j}} + C_{i,j}. \tag{4.1}$$

Because $\tau_{i,j}$ is a first order approximation, the formula is valid only if $C_{i,j} \ll \mu_{i,j}$. When a fault strikes, there is first a downtime of duration D, and then a recovery period of duration $R_{i,j}$. We assume that $R_{i,j} = C_{i,j}$, while the downtime value D is platform-dependent and not application-dependent.

4.2.2 Execution time without redistribution

To compute the expected execution time of a schedule, we first have to compute the expected execution time of an application T_i executed on j processors subject to failures. We first consider the case without redistribution (but taking failures into account). Let $t_{i,j}$ be the execution time of application T_i on j processors in a fault-free scenario. Let $t^R_{i,j}(\alpha)$ be the expected time required to compute a fraction α of the total work for application T_i on j processors, with $0 \leq \alpha \leq 1$. We need to consider such a partial execution of T_i on j processors to prepare for the case with redistributions.

Recall that the execution of application T_i is periodic, and that the period $\tau_{i,j}$ depends only on the number of processors, but not on the remaining execution time (see Equation 4.1). After a work of

duration $\tau_{i,j} - C_{i,j}$, there is a checkpoint of duration $C_{i,j}$. In a fault-free execution, the time required to execute the fraction of work α is $\alpha t_{i,j}$, hence a total number of checkpoints of

$$N_{i,j}^{\text{ff}}(\alpha) = \left\lfloor \frac{\alpha t_{i,j}}{\tau_{i,j} - C_{i,j}} \right\rfloor. \tag{4.2}$$

Next, we have to estimate the expected execution time for each period of work between checkpoints. We are able to calculate the expectation of one period of work according to an MTBF value and a number of processors. The expected time to execute successfully during T units of time with j processors (there are $T - C$ units of work and C units of checkpoint, where T is the period) is equal to $\left(\frac{1}{\lambda j} + D\right)(e^{\lambda j T} - 1)$ [58]. Therefore, in order to compute $t_{i,j}^R(\alpha)$, we compute the sum of the expected time for each period, plus the expected time for the last (possibly incomplete) period. This last period is denoted as $\tau_{last}(\alpha)$ and defined as $\tau_{last}(\alpha) = \alpha t_{i,j} - N_{i,j}^{\text{ff}}(\alpha)(\tau_{i,j} - C_{i,j})$.

The first $N_{i,j}^{\text{ff}}(\alpha)$ periods are equal (of length $\tau_{i,j}$), hence have the same expected time. Finally, we obtain:

$$t_{i,j}^R(\alpha) = e^{\lambda j R_{i,j}}\left(\frac{1}{\lambda j} + D\right)\left(N_{i,j}^{\text{ff}}(\alpha)(e^{\lambda j \tau_{i,j}} - 1) + (e^{\lambda j \tau_{last}(\alpha)} - 1)\right). \tag{4.3}$$

In a fault-free environment, it is natural to assume that the execution time is non-increasing with the number of processors. Here, this assumption would translate into the condition:

$$t_{i,j+1}^R(\alpha) \leq t_{i,j}^R(\alpha) \text{ for } 1 \leq i \leq n,\ 1 \leq j < p,\ 0 \leq \alpha \leq 1. \tag{4.4}$$

However, when we allocate more processors to an application, even though the work is further parallelized, the probability of failures increases, and the corresponding waste increases as well. Therefore, adding resources to an application is useful up to a threshold. After this threshold, we have $t_{i,j+1}^R \geq t_{i,j}^R$. In order to satisfy Equation 4.4, we restrict the number of processors assigned to each application, and never assign more processors than the previous threshold. In other words, if T_i is already assigned j processors, we consider assigning more processors to it only if $t_{i,j+1}^R \leq t_{i,j}^R$. Formally, this defines a maximum number of processors, $j_{max}(i)$, for each application T_i:

$$j_{max}(i) = \min_{1 \leq j \leq p} \{j \text{ such that } t_{i,k}^R \geq t_{i,j}^R \text{ for all } k > j\}, \tag{4.5}$$

and we assume that $t_{i,j+1}^R \leq t_{i,j}^R$ for all $j < j_{max}(i)$.

Another common assumption for malleable applications is that the work is non-decreasing when the number of processors increases [18]: this amounts to say that no super-linear speed-up is possible. Hence, we assume here that for $1 \leq i \leq n$, $1 \leq j < p$ and $0 \leq \alpha \leq 1$, $(j+1) \times t_{i,j+1}^R(\alpha) \geq j \times t_{i,j}^R(\alpha)$.

For convenience, we denote by t_i^U the current expected finish time of application T_i at any point of the execution. Initially, if application T_i is allocated to j processors, we have $t_i^U = t_{i,j}^R(1)$.

4.2.3 Redistributing processors

There are two major cases for which it may be useful to redistribute processors: (1) in a fault-free scenario, when an application ends, it releases processors that can be used to accelerate other applications, and (2) when an error strikes, we may want to force the release of processors, so that we can assign more processors to the application that has been slowed down by the error. We first consider a fault-free scenario, and then we account for the checkpoint costs and for redistribution after failures. Finally, we discuss the case of consecutive redistributions.

Fault-free scenario

We first consider a simplified scenario without checkpoint (nor failure), in order to explain how redistribution works. Consider for instance that q processors are released when application T_2 ends. We can allocate q_1 new processors to application T_1, and q_3 new processors to application T_3, where $q_1 + q_3 = q$ (see Figure 4.2). This redistribution will take some time (redistribution cost RC_i, detailed below), after which T_1 and T_3 will resume execution, and we first need to compute the new expected completion time for their remaining fraction of work.

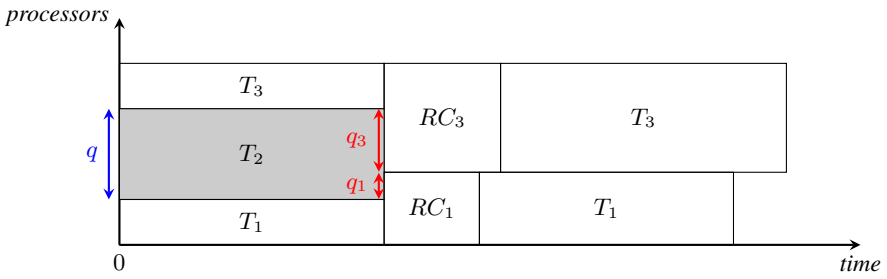


Figure 4.2: Redistribution at the end of an application, where RC_i represents the redistribution cost for task T_i.

Consider that a redistribution is conducted at time t_e (the end time of an application), and that application T_i, initially with j processors, now has $k = j+q > j$ processors. What will be the new finish time of T_i? The fraction of work already executed for T_i is $\frac{t_e}{t_{i,j}}$, because the application was supposed to finish at time $t_{i,j}$ (see Figure 4.3). The remaining fraction of work is $\alpha = 1 - \frac{t_e}{t_{i,j}}$, and the time required to complete this work with k processors is t', where $\frac{t'}{t_{i,k}} = \alpha$, hence $t' = \alpha t_{i,k} = \left(1 - \frac{t_e}{t_{i,j}}\right) t_{i,k}$.

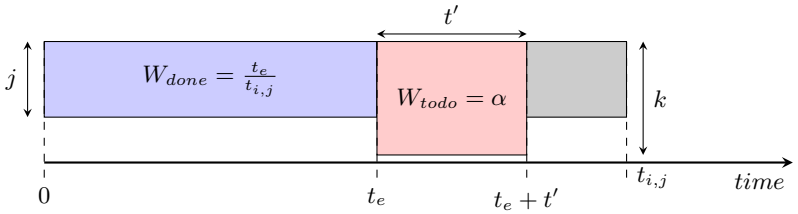


Figure 4.3: Work representation for application T_i at time t_e.

Furthermore, we need to add a redistribution cost: when moving from j to $k = j + q$ processors, the application T_i must redistribute its data across the processors. The application keeps its initial j processors, which now hold too much data, and enrolls $q = k - j$ new processors, which have no data yet. Recall that m_i is the memory footprint (total data size) of application T_i. Each of the original j processors initially holds $\frac{m_i}{j}$ data and will keep only $\frac{m_i}{k}$ after the redistribution; it sends $\frac{m_i}{jk}$ data to each of the newly enrolled q processors, thereby keeping $\frac{m_i}{j} - (k-j)\frac{m_i}{jk} = \frac{m_i}{k}$ data. In turn, each new processor receives $\frac{m_i}{jk}$ data from j processors and duly gets $\frac{m_i}{k}$ data in the end.

What is the best schedule for such a redistribution, and what time does it require? We first account for a constant start-up overhead S, paid for initiating the redistribution call. Then we adopt a realistic one-port communication model [14] where a processor can send and receive at most one message at

any time-step. Independent communications, involving distinct sender/receiver pairs, can take place in parallel: however, two messages sent by the same processor will be serialized. Recall that the time to communicate a message of size s is $\beta + \frac{s}{\tau}$. To schedule the redistribution, we build a bipartite graph G with j nodes on the left and q nodes on the right. In the one-port model, there can be up to j simultaneous communications (each of size $\frac{m_i}{jk}$) involving j distinct processor pairs. Let us call a *round* such a set of simultaneous (independent) communications. How many rounds are required to schedule the redistribution? We transform this problem into an edge coloring problem, with one color for one round (see Figure 4.4). The number of rounds required is equal to the edge chromatic number $\chi'(G)$. Konig's theorem [19] states that this edge chromatic number is equal to the maximum degree in G so $\chi'(G) = \Delta(G)$ when G is bipartite. Clearly, we have here $\Delta(G) = \max(j, k-j)$. Therefore, the number of rounds is equal to $\max(j, k-j)$, and the redistribution cost is $RC_i^{j\to k} = S + \max(j, k-j) \times \left(\frac{m_i}{jk\tau} + \beta\right)$.

Needless to say, we would perform a redistribution if the cost of redistribution is lower than the benefit of allocating new processors to the application, i.e., if $t_{i,j} - (t_e + t') > RC_i^{j\to k}$.

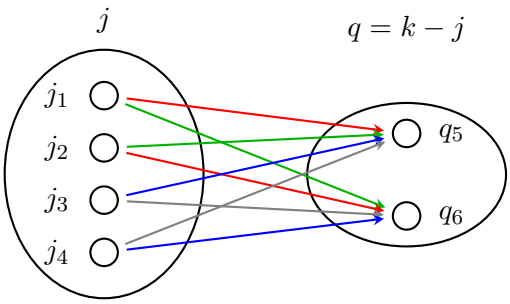


Figure 4.4: Bipartite graph G representing a redistribution from $j = 4$ to $k = 6$ processors, with each communication round colored. We have $\chi'(G) = \Delta(G) = 4$.

Accounting for failures

When struck by a fault, an application needs to recover from the failure and to re-execute some work. While the application loads were well-balanced initially in order to minimize total execution time, now the faulty application is likely to exceed its expected execution time. If it becomes the longest application of the schedule, we try to assign it more processors so as to reduce its completion time, hence redistributing processors.

Because we use the double checkpointing algorithm as resilience model, we consider processors by pairs. We aim at redistributing pairs of processors either when an application is finished, at time t_e (as in the fault-free scenario discussed in Section 4.2.3), or when a failure occurs, say at time t_f. In each case, we need to compute the remaining work, and the new expected completion time of the applications that have been affected by the event. Given an application T_i, we keep track of the time when the last redistribution or failure occurred for this application, denoted as t_{lastR_i}. At time t (corresponding to the end of an application or to a failure), we know exactly how many checkpoints have been taken by application T_i executed on j processors since t_{lastR_i}, and we let this number be $N_{i,j}$:

$$N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor. \tag{4.6}$$

We begin with the case of an application completion: consider that an application finishes its execution at time t_e, hence releasing some processors. We consider assigning some of these processors to an application T_i currently running on j processors. The fraction of work executed by T_i since the last redistribution is $\frac{t_e - t_{lastR_i} - N_{i,j}C_{i,j}}{t_{i,j}}$, because we have to remove the cost of the checkpoints, during which the application did not execute useful work.

We apply the same reasoning for the second case, when a fault occurs. In this case, we need to consider the application T_i where the failure stroke, and other applications $T_{i'}$ from which we would remove some processors (in order to give them to T_i).

- Consider that application T_i is running on j processors and subject to a failure at time t_f. Therefore, T_i needs to recover from its last valid checkpoint, and the fraction of work executed by T_i corresponds to the number of entire periods completed since the last failure or redistribution t_{lastR_i}, each followed by a checkpoint. We can express it as $\frac{N_{i,j} \times (\tau_{i,j} - C_{i,j})}{t_{i,j}}$.

- At time t_f, consider application $T_{i'}$, on which we perform a redistribution, moving from j' to $j' - q$ processors, with $q > 0$. The fraction of work executed by $T_{i'}$ can be computed as in the application ending case scenario: it is $\frac{t_f - t_{lastR_{i'}} - N_{i',j'}C_{i',j'}}{t_{i',j'}}$.

Finally, for any application subject to a redistribution or a failure, let α_i be the remaining fraction of work to be executed by T_i, that is 1 minus the sum of the fraction of work executed before t_{lastR_i} and the fraction of work expressed above (computed between t_{lastR_i} and t).

Similarly to the fault-free scenario, $RC_i^{j \to k}$ denotes the redistribution cost for application T_i when moving from j to k processors. Redistribution can now add ($k > j$) or remove ($k < j$) processors to application T_i, and the cost is expressed as:

$$RC_i^{j \to k} = S + \max(\min(j,k), |k-j|) \times \left(\frac{m_i}{kj\tau} + \beta \right). \tag{4.7}$$

We are now ready to compute the new values of t_{lastR_i} for all applications subject to a failure or a redistribution, and we illustrate the different scenarios in Figure 4.5. Let t be the time of the event (end of application $t = t_e$, or failure $t = t_f$), and consider that a redistribution is done either for a faulty application T_i or for another application $T_{i'}$. After a redistribution, we always start by taking a checkpoint before computing with the new period. Therefore, if a fault occurs, we do not have to redistribute again.

For the faulty application T_i, the new value of t_{lastR_i} hence becomes $t_{lastR_i} = t + D + R_{i,j} + RC_i^{j \to k} + C_{i,k}$ (we need to account for the downtime and recovery). However, if $T_{i'}$ is performing a redistribution but it was not struck by a failure, it can start the redistribution at time t: either it is getting new processors that are available following the end of an application, or is is using less processors and can perform its redistribution. In all cases, we have $t_{lastR_{i'}} = t + RC_{i'}^{j' \to k'} + C_{i',k'}$. Note that we can have processors involved simultaneously in two redistributions, as they will only receive data from the other processors of the faulty application T_i, and send data to the other processors of the non-faulty application $T_{i'}$. We assume that sends and receives can be done in parallel without slowdown.

Finally, the expected finish time of an application T_i for which we have updated t_{lastR_i} becomes $t_i^U = t_{lastR_i} + t_{i,k}^R(\alpha_i)$, where k is the new number of processors on which T_i is executed, and α_i the remaining fraction of work. Similarly to the fault-free scenario, we give extra processors to an application only if the new expected finish time t_i^U is lower than the one with no redistribution.

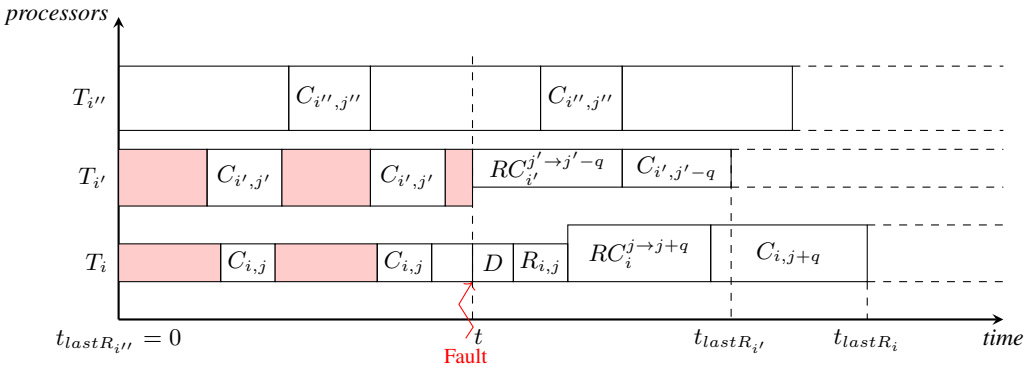


Figure 4.5: Example of redistribution when a fault strikes application T_i. The colored rectangles correspond to useful work done by T_i and $T_{i'}$ before the failure. $T_{i''}$ is not affected by the failure as it does not perform a redistribution.

When multiple redistributions overlap

Here, we deal with the problem of chaining redistributions. If another event (application completion or fault) occurs during the current redistribution, we cannot enroll the processors that have not yet finished the current redistribution. On Figure 4.6, at the end of the application T_3, there are no available applications to whom we may try to give its processors. T_1 will be able to start a new redistribution at time-step t_1, and T_2 at time-step r_2.

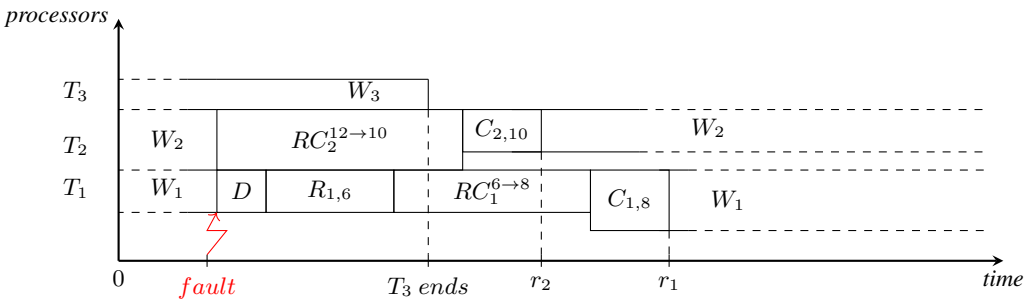


Figure 4.6: Illustration of two consecutive events.

4.2.4 Objective function

We can now state the objective function: Given n malleable applications $\{T_1, \ldots, T_n\}$, their speedup profiles, and an execution platform with p identical processors subject to failures with individual rate λ, CoSched aims at minimizing the maximum of the expected completion times of the applications. Redistributions are allowed only when an application completes execution or is struck by a failure (with a cost specified in Section 4.2.3).

4.3 Complexity results

We first consider the CoSched problem without redistributions and provide an optimal polynomial-time algorithm. Then, we prove that the problem becomes NP-complete with redistributions, even in a fault-free scenario.

4.3.1 Without redistributions

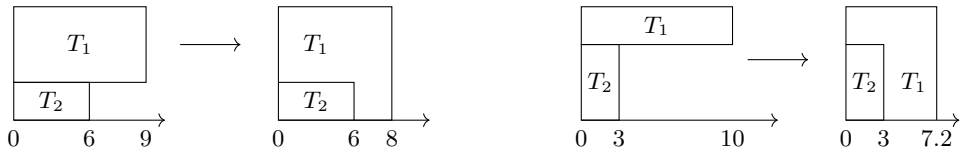
Aupy et al. [9] designed a greedy algorithm to solve the problem with no redistribution, in a fault-free scenario. Their algorithm (called Optimal-1-pack-schedule) therefore works with $t_{i,j}$ values instead of $t^R_{i,j}$, and minimizes the execution time of the applications. As a minor detail, it does not take into account the fact that the number of processors assigned to an application must always be even in our setting, because we use the double checkpointing algorithm. It is not difficult to extend this algorithm to solve the problem with failures, but still without redistributions: the idea is to give initially two processors per applications, to sort them by expected execution time, and to greedily give two extra processors to the longest application, if it decreases its expected execution time. This algorithm is called Opt-NoRedistrib. We can therefore prove the following theorem.

Theorem 4.1. *The* CoSched *problem without redistributions can be solved in polynomial time $O(n)$, where n is the number of applications.*

Proof. We define a function σ such that $\sum_{i=1}^{n} \sigma(i) \leq p$, where $\sigma(i)$ is the number of processors assigned to T_i. A schedule with no redistribution corresponds to a unique function σ, because the number of processors remains identical throughout the whole execution. The fraction of work that each application must compute is $\alpha = 1$, and we use the notation $T_i \preccurlyeq^R_\sigma T_j$ if $t^R_{i,\sigma(i)}(1) \leq t^R_{j,\sigma(j)}(1)$. Then, Algorithm 5 returns in polynomial time a schedule that minimizes the expected execution time. It greedily allocates processors to the longest application while its expected execution time can be decreased. If we cannot decrease the expected execution time of the longest application, then we cannot decrease the overall expected execution time, which is the maximum of the expected execution times of all applications.

Algorithm 5: Optimal schedule with no redistribution.

```
1  procedure Opt-NoRedistrib (n, p) begin
2      for i = 1 to n do σ(i) := 2 ;
3      Let L be the list of applications sorted in non-increasing values of ≼^R_σ;
4      p_available := p − 2n;
5      while p_available ≥ 2 do
6          T_i* := head(L);
7          L := tail(L);
8          if σ(i*) < j_max(i*) then
9              σ(i*) := σ(i*) + 2;
10             L := Insert T_i* in L according to its ≼^R_σ value;
11             p_available := p_available − 2;
12         else p_available := 0;
13     end
14     return σ;
15 end
```



(a) OPT-NOREDISTRIB uses largest execution time to allocate processors

(b) GREEDYSPEEDUPPROFILE uses best speedup profile to allocate processors

Figure 4.7: Examples: coordinates are execution time (x-axis) and processors (y-axis).

The proof that this algorithm returns an optimal cost schedule is similar to the proof in [9]. We replace $t_{i,j}$ by $t^R_{i,j}(1)$, and instead of adding processors one-by-one, we add them two-by-two. Consequently, there are at most $(p-2n)/2$ iterations. The complexity of Algorithm 5 is $O(p \times \log(n))$.

□

Note that we added a test in Line 8 to check whether there is a hope to decrease the expected execution time of the longest application. If $T_{i^\star}$ has reached its maximum enrollment with $\sigma(i^\star)$ processors (according to the threshold $j_{max}(i^\star)$ defined with Equation 4.5), then we cannot decrease its expected execution time. In the following, in such situations, we aim at making good use of extra processors through redistributions.

4.3.2 With redistributions

We show through a few examples the difficulty of COSCHED when redistributions are allowed, even when there are no failures. The first example shows that the previous algorithm OPT-NOREDISTRIB is no longer optimal. Consider two applications T_1 and T_2 and three processors, and further assume that there is no cost for redistribution. We use the following speedup profiles:

$$T_1 = \begin{cases} t_{1,1} = 10, & w_{1,1} = 10 \\ t_{1,2} = 9, & w_{1,2} = 18 \\ t_{1,3} = 6, & w_{1,3} = 18 \end{cases} \qquad T_2 = \begin{cases} t_{2,1} = 6, & w_{2,1} = 6 \\ t_{2,2} = 3, & w_{2,2} = 6 \end{cases}$$

where $w_{i,j}$ represents the work for application i with j processors, i.e., $w_{i,j} = j \times t_{i,j}$.

OPT-NOREDISTRIB initially assigns one processor to each application, and then the remaining one to the longest application T_1. At time 6, when T_2 finishes and releases its processor, we redistribute T_1 over the three processors. At time 6, the application T_1 has done $2/3$ of its work, it remains $1/3 \times t_{1,3} = 1/3 \times 6 = 2$ time units with 3 processors, therefore T_1 ends at time $6 + 2 = 8$ (see Figure 4.7a). We obtain a smaller makespan if we do not use OPT-NOREDISTRIB but instead the variant GREEDYSPEEDUPPROFILE, where remaining processors are allocated to the application with the best speedup profile. In the example, GREEDYSPEEDUPPROFILE initially allocates the third processor to T_2 because the execution time with two processors is divided by two, i.e., perfect speedup with $w_{2,2}/w_{2,1} = 1$. Then T_2 finishes at time 3. At this time, T_1 has still to complete $7/10$ of its load, so the remaining time for T_1 is equal to $7/10 \times t_{1,3} = 7/10 \times 6 = 4.2$. The makespan in this second configuration becomes $3 + 4.2 = 7.2$, which is better!

Since OPT-NOREDISTRIB is no longer optimal, a natural question is whether GREEDYSPEEDUPPROFILE is optimal. The following example answers negatively. Consider the following speedup profiles

(with two applications and three processors as before):

$$T_1 = \begin{cases} t_{1,1} = 10, & w_{1,1} = 10 \\ t_{1,2} = 6, & w_{1,2} = 12 \\ t_{1,3} = 5, & w_{1,3} = 15 \end{cases} \qquad T_2 = \begin{cases} t_{2,1} = 6, & w_{2,1} = 6 \\ t_{2,2} = 3, & w_{2,2} = 6 \end{cases}$$

GREEDYSPEEDUPPROFILE allocates two processors to T_2 (best speedup profile) and one processor to T_1. So at time 3, the application T_2 completes and its two processors are given to T_1. The execution time for T_1 is $3 + 7/10 \times 5 = 6.5$. But if we allocate two processors to T_1 and one to T_2, we finish both applications at time 6 without any redistribution!

Intuitively, these little examples show that COSCHED seems to be of combinatorial nature when redistributions are taken into account, even with zero cost.

To establish the complexity of the problem with redistributions, we consider the simple case with no failures. Therefore, redistributions occur only at the end of an application, and any application changes at most n times its number of processors, where n is the total number of applications. We further consider that the redistribution cost is a constant equal to S, i.e., we let $\beta = 0$ and $\tau = +\infty$ in Equation 4.7. Even in this simplified scenario, the problem is NP-complete:

Theorem 4.2. *With constant redistribution costs and without failures,* COSCHED *is NP-complete (in the strong sense).*

Proof. We consider the associated decision problem: given a bound on the execution time D, is there a schedule whose expected execution time does not exceed D? The problem is obviously in NP: with n applications, there are at most $n - 1$ redistributions, hence n intervals during which processor assignment remains constant for all applications. Given a schedule and the list of resources assigned to each application within these n intervals, it is easy to check in polynomial time that it is valid and that its execution time does not exceed the bound D.

To establish the completeness, we use a reduction from 3-PARTITION [48] with distinct integers (which still remains strongly NP-complete [59, Corollary 7]). We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3m$ distinct positive integers $a_1, a_2, \ldots, a_{3m}$ such that for all $i \in \{1, \ldots, 3m\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{3m} a_i = mB$, does there exist a partition $I_1, \ldots, I_m$ of $\{1, \ldots, 3m\}$ such that for all $j \in \{1, \ldots, m\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? Letting $M = \max_{1 \leq i \leq 3m}(a_i)$, we can assume w.lo.g. that $B \leq 3M$, otherwise there is no solution to $\mathcal{I}_1$.

We build an instance $\mathcal{I}_2$ of our problem, with $n = 4m$ applications and $p = n$ processors. We let $D = 3M + 2$ be the bound on the execution time. For each redistribution, each application whose processor number changes, simply pays the constant overhead $S = \frac{1}{9m} < 1$ (communication costs are set to zero). For $1 \leq i \leq 3m$, we have the following execution times: $t_{i,1} = a_i$, and $t_{i,j} = \frac{3a_i}{4}$ for $j > 1$ (these are *small* applications, and the work is strictly larger when using more than one processor). The last m applications are identical, with the following execution times: for $3m + 1 \leq i \leq 4m$, $t_{i,j} = \frac{4D - B - 9S}{j}$ for $1 \leq j \leq 4$, and $t_{i,j} = \frac{2}{9}(4D - B - 9S)$ for $j > 4$ (these are *large* applications with a total work equal to $4D - B - 9S$ for $1 \leq j \leq 4$, and a strictly larger work when using more than four processors). It is easy to check that the execution times are non-increasing with j, and that the work $j \times t_{i,j}$ is non-decreasing with j for all applications. Note that $4D - B - 9S > D$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. Let $I_k = \{a'_{1,k}, a'_{2,k}, a'_{3,k}\}$, for $k \in \{1, \ldots, m\}$. We build the following schedule for $\mathcal{I}_2$: initially, each application has a single processor. When an application T_i finishes its execution (at time a_i), with $1 \leq i \leq 3m$, its processor is redistributed to application T_{3m+k}, given that $a_i \in I_k$. Both the single processor of T_i and each currently enrolled processor of

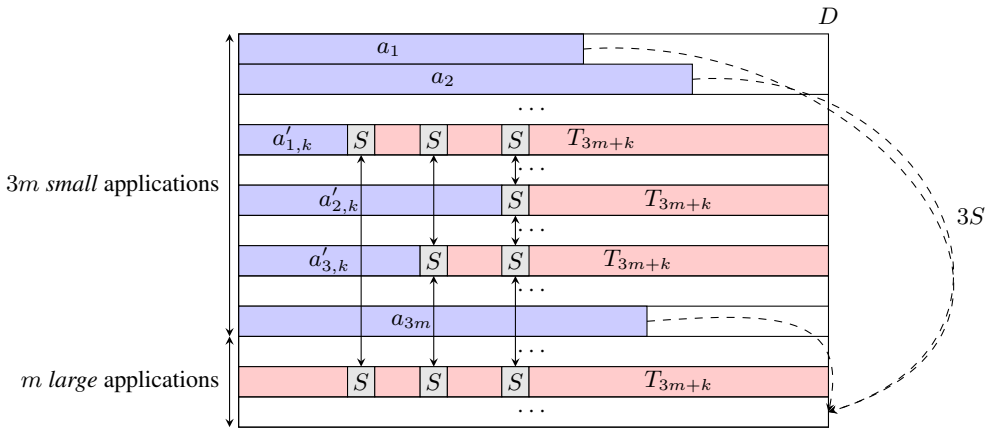


Figure 4.8: Illustration for the proof of Theorem 4.2.

T_{3m+k} pay a time overhead S for this redistribution, see Figure 4.8 for an illustration. Because the a_i's are all distinct, the successive redistributions occur at different time-steps, and the redistribution intervals of size S do not overlap. Each application T_{3m+k} starts with 1 processor and proceeds first with 2 processors (then paying an overhead S for its single processor before the redistribution), then with 3 processors (then paying an overhead S for each of its two processors before the redistribution), and finally with 4 processors (then paying an overhead S for each of its three processors before the redistribution) for some time in the end of its execution, because $M + S < D$. The total overhead due to the redistributions involving the three small tasks giving resources to T_{3m+k} is therefore $9S$. Now, each application T_{3m+k} always executes with an optimal work profile, and actually completes its execution in time D. Indeed, the 4 processors finally assigned to T_{3m+k} have to complete a total work of $a'_{1,k} + a'_{2,k} + a'_{3,k} + 4D - B - 9S = 4D - 9S$, and there are exactly $3(D - S) + D - 6S = 4D - 9S$ time slots available for computations. Again, because $M + S < D$, all small applications also complete before the deadline, and we have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Initially, we have one processor per application, because there are exactly n processors and n applications. We first show that each small application T_i terminates before the end of the schedule, and that its processor must be redistributed. Indeed, $a_i \leq M < D$, and if we do not redistribute the processor assigned to T_i when it completes, then this processor stays idle for $D - a_i > D - M$ time steps. But the total work to execute is at least $\sum_{i=1}^{3m} a_i + m \times (4D - B - 9S) = m(4D - 9S)$, assuming perfect parallelism. If the remaining $n - 1$ processors work all the time, they contribute for $(n - 1)D$. If the processor assigned to T_i works at most M time-steps, we must have $m(4D - 9S) \leq (n - 1)D + M$, or equivalently $9mS \geq D - M$. But $D - M > 2$, and $9mS \leq 1$ by definition of S, a contradiction.

Because the a_i's are distinct, the $3m$ redistributions at the end of the $3m$ small tasks do not overlap. The first m redistributions involve at least another application running on one processor, which also looses S time-steps. The next m redistributions involve at least another application running on two processors, which costs $2S$ work units, and finally the last m redistributions involve at least another application running on three processors, hence costing $3S$ work units. Altogether, we have at least $9mS$ work units for redistribution costs. But the total work is at least $nD - 9mS$, and the area of the computing window is nD. This means that we pay exactly $9mS$ for redistributions, and that all the work is perfectly parallel. We now draw two consequences:

- When a small task completes, the redistribution of its processor involves a single other application (otherwise we would end with strictly more than $9mS$ redistribution overhead).
- This processor is redistributed to a large application, because all the work is perfectly parallel.

There are $3m$ processors to redistribute to m large applications, and none of them can receive more than 3 processors, again because all the work is perfectly parallel. Hence, each large application is assigned exactly 3 new processors throughout its execution. Formally, for $1 \leq k \leq m$, the large application T_{3m+k} receives processors from 3 small applications T_i with $i \in I_k = \{a'_{1,k}, a'_{2,k}, a'_{3,k}\}$, for $k \in \{1, \ldots, m\}$. The total work of these four processors is $4D - B - 9S + a'_{1,k} + a'_{2,k} + a'_{3,k}$ and there are $4D - 9S$ available time-steps for them. Hence $a'_{1,k} + a'_{2,k} + a'_{3,k} \leq B$. This is true for all triplets of small applications, and because $\sum_{i=1}^{3m} a_i = mB$, we must have an equality for each triplet, hence the solution to $\mathcal{I}_1$. □

We conjecture that CoSched remains NP-complete with zero redistribution cost. This is because of the combinatorial exploration suggested by the examples. But this remains an open problem!

4.4 Heuristics

In this section, we introduce polynomial-time heuristics to solve the general CoSched problem with both failures and redistributions. Before performing any redistribution, we need to choose an initial allocation of the p processors to the n applications. We use the optimal algorithm without redistribution discussed in Section 4.3 (Opt-NoRedistrib).

We first discuss the general structure of the heuristics. Then, we explain how to redistribute available processors, and the two strategies to redistribute when failures occur.

4.4.1 General structure

All heuristics share the same skeleton (see Algorithm 6): we iterate over each event (either a failure or an application termination) until total remaining work is equal to zero. If some applications are still working for a previous redistribution, (i.e., the current time t is smaller than t_{lastR_i} for these applications), then we exclude them for the next redistribution (Line 15), and add them back into the list of applications after the current redistribution is completed. If an application ends, we redistribute available processors as will be discussed in Section 4.4.2. Then, if there is a failure, we calculate the new expected execution time of the faulty application (Line 26). Also, we remove from the list the applications that end before t_{lastR_f}, and we release their processors (Line 28).

Afterwards, we have to choose between trying to redistribute or do nothing. If the faulty application is not the longest application, the total execution time has not changed since the last redistribution. Therefore, because it is the best execution time that we could reach, there is no need to try to improve it. However, if the faulty application is the longest application (Line 30), we apply a heuristic to redistribute processors (see below).

4.4.2 Redistribution when an application ends

When an application ends, the idea is to redistribute the processors that it releases in order to decrease the expected execution time. The easiest way to proceed consists in adding processors greedily to the application with the longest execution time, as was done in Opt-NoRedistrib to compute an optimal schedule. This time, we further account for the redistribution cost, and update the values of α_i,

Algorithm 6: Algorithmic skeleton

```
1  procedure Main(n, p)
2  begin
3      α and t_lastR are considered as global variables;
4      /* Initial schedule */;
5      σ := OPT-NOREDISTRIB (n, p);
6      for i = 1 to n do
7          α_i = 1; t_lastR_i = 0;
8          t_i^U = t^R_{i,σ(i)}(1);
9      end
10     Let L be the list of applications sorted in non-increasing values of t_i^U;
11     /* While it remains work */;
12     while Σ_{i=1}^n α_i > 0 do
13         k := p − Σ_{i=1}^n σ(i) /* There are k unused processors */;
14         t := next incoming event;
15         for i = 1 to n do if t ≤ t_lastR_i then Remove temporarily T_i from L;
16         ;
17         if t is the end of application T_e then
18             α_e := 0 ;
19             Remove T_e from the list of applications L;
20             σ := REDISTRIB-AVAILABLE-PROCS(L, t, k + σ(e), σ);
21         else if t is a failure striking application T_f then
22             /* Updating information about the faulty application T_f */
23             j := σ(f); N_{f,j} = ⌊(t − t_lastR_f)/τ_{f,j}⌋;
24             α_f := α_f − N_{f,j}(τ_{f,j} − C_{f,j})/t_{f,j};
25             t_lastR_f := t + D + R_{f,j};
26             t_f^U := t_lastR_f + t^R_{f,j}(α_f);
27             Update the position of T_f in the list L according to its new t_f^U value;
28             for i = 1 to n do if T_i finishes before t_lastR_f then Remove T_i and release σ(i)
                 processors;
29             ;
30             if t_f^U = max_{1≤i≤n} t_i^U then
31                 σ := APPLY-HEURISTIC(L, t, f, σ);
32             end
33         end
34         ;
35         Put back the previously removed applications into L;
36     end
37 end
```

t_{lastR_i} and t_i^U for each application i that encountered a redistribution. Therefore, this heuristic, called EndLocal (see Algorithm 7), returns a new distribution of processors.

Algorithm 7: EndLocal

```
1  procedure EndLocal (L, t, k, σ)
2  begin
3      σ_init := σ;
4      while k ≥ 2 do
5          T_i := head(L); L := tail(L);
6          j := σ_init(i);
7          N_{i,j} = ⌊(t − t_{lastR_i})/τ_{i,j}⌋;
8          α_i^t := α_i − (t − t_{lastR_i} − N_{i,j}C_{i,j})/t_{i,j};
9          /* We first check whether T_i can be improved */
10         if σ(i) < j_max(i) then
11             σ(i) := σ(i) + 2;
12             t_i^U := t + RC_i^{j→σ(i)} + C_{i,σ(i)} + t^R_{i,σ(i)}(α_i^t);
13             L := Insert T_i in L according to its t_i^U value;
14             k := k − 2;
15         end
16     end
17     /* Updating α_i and t_{lastR_i} if needed */
18     for i = 1 to n do
19         j := σ_init(i);
20         if σ(i) ≠ j then
21             N_{i,j} = ⌊(t − t_{lastR_i})/τ_{i,j}⌋;
22             α_i := α_i − (t − t_{lastR_i} − N_{i,j}C_{i,j})/t_{i,j};
23             t_{lastR_i} := t + RC_i^{j→σ(i)} + C_{i,σ(i)};
24         end
25     end
26     return σ;
27 end
```

Rather than using only local decisions to redistribute available processors at time t, it is possible to recompute an entirely new schedule, using Opt-NoRedistrib again, but further accounting for the cost of redistributions. This heuristic is called EndGreedy (see Algorithm 8). Now, we need to compute the remaining fraction of work for each application, and we obtain an estimation of the expected finish time when each application is mapped on two processors. Similarly to Opt-NoRedistrib, we then add two processors to the longest application while we can improve it, accounting for redistribution costs.

Note that we effectively update the values of α_i and t_{lastR_i} for application T_i only if a redistribution was conducted for this application. It may happen that the algorithm assigns the same number of processors as was used before. Therefore, we keep the updated value of the fraction of work in a temporary variable α_i^t and update it whenever needed at the end of the procedure.

Algorithm 8: ENDGREEDY

procedure ENDGREEDY (L, t, k, σ)
begin
  $\sigma_{init} := \sigma$;
  for $i = 1$ **to** n **do**
    $N_{i,j} = \lfloor (t - t_{lastR_i})/\tau_{i,j} \rfloor$;
    $\alpha_i^t := \alpha_i - (t - t_{lastR_i} - N_{i,j}C_{i,j})/t_{i,j}$;
    $\sigma(i) \leftarrow 2$;
    if $\sigma(i) \neq \sigma_{init}(i)$ **then** $t_i^U = t + RC_i^{\sigma_{init} \rightarrow \sigma(i)} + C_{i,\sigma(i)} + t_{i,\sigma(i)}^R(\alpha_i^t)$;
  end
  Let L be the list of applications sorted in non-increasing values of t_i^U;
  $p_{available} := p - 2n$;
  while $p_{available} \geq 2$ **do**
    $T_i := head(L)$; $L := tail(L)$;
    $improvable := false$; $q := 2$;
    while $\sigma(i) + q < j_{max}(i)$ **do**
      if $\sigma(i) + q = \sigma_{init}(i)$ **then** $t^E := t_{lastR_i} + t_{i,\sigma(i)+q}^R(\alpha_i)$;
      else $t^E := t + t_{i,\sigma(i)+q}^R(\alpha_i^t) + RC_i^{\sigma_{init}(i) \rightarrow \sigma(i)+q} + C_{i,\sigma(i)+q}$;
      if $t^E < t_i^U$ **then** $improvable := true$; $q := j_{max}(i)$;
      else $q := q + 2$;
    end
    if $improvable$ **then**
      $\sigma(i) := \sigma(i) + 2$;
      if $\sigma(i) = \sigma_{init}(i)$ **then** $t_i^U := t_{lastR_i} + t_{i,\sigma(i)}^R(\alpha_i)$;
      else
        $t_i^U := t + RC_i^{\sigma_{init} \rightarrow \sigma(i)} + C_{i,\sigma(i)} + t_{i,\sigma(i)}^R(\alpha_i^t)$;
      end
      $L :=$ Insert T_i in L according to its t_i^U value;
      $p_{available} := p_{available} - 2$;
    end
    else $p_{available} := 0$;
  end
  /* Updating t_{lastR_i} and α_i if needed */
  for $i = 1$ **to** n **do**
    if $\sigma(i) \neq \sigma_{init}(i)$ **then**
      $\alpha_i := \alpha_i^t$;
      $t_{lastR_i} := t + RC_i^{\sigma_{init}(i) \rightarrow \sigma(i)} + C_{i,\sigma(i)}$;
    end
  end
  return σ;
end

4.4.3 Redistribution when there is a failure

Similarly to the case of an application ending, we propose two heuristics to redistribute in case of failures. The first one, SHORTESTAPPLICATIONSFIRST, takes only local decisions. First, we allocate the k available processors (if any) to the faulty application if that application is improvable. Then, if the faulty application is still improvable, we try to take processors from shortest applications (denoted T_s) in the schedule, and give these processors to the faulty application, until the faulty application is no longer improvable, or there are no more processors to take from other applications. We take processors from an application only if its new execution time is smaller than the execution time of the faulty application (see Algorithm 9).

The second heuristic, ITERATEDGREEDY, uses a modified version of the greedy algorithm that initializes the schedule (OPT-NOREDISTRIB) each time there is a failure, while accounting for the cost of redistributions. This is done similarly to the redistribution of ENDGREEDY explained above, except that we need to handle the faulty application differently to update the values of α_f and t_{lastR_f} (see Algorithm 10).

4.5 Simulations

To assess the efficiency of the heuristics defined in Section 4.4, we have performed extensive simulations. The simulation settings are discussed in Section 4.5.1, and results are presented in Section 4.5.2. Note that the code is publicly available at `http://graal.ens-lyon.fr/~abenoit/code/redistrib`, so that interested readers can experiment with their own parameters.

4.5.1 Simulation settings

To evaluate the quality of the heuristics, we conduct several simulations, using realistic parameters. The first step is to generate a fault distribution: we use an existing fault simulator developed in [20, 23]. In our case, we use this simulator with an exponential law of parameter λ. The second step is to generate a fault-free execution time for each application (the $t_{i,j}$ value). We use a *synthetic* model to generate the execution profiles in order to represent a large set of scientific applications. The application model that we use is a classical one, similar to the one used in [9]. For a problem of size m, we define the sequential time: $t(m,1) = 2 \times m \times log_2(m)$. Then we can define the parallel execution time on q processors:

$$t(m,q) = f \times t(m,1) + (1-f)\frac{t(m,1)}{q} + \frac{m}{q} log_2(m). \tag{4.8}$$

The parameter f is the sequential fraction of time, we fix it to $f = 0.08$. So 92% of time is considered as parallel. The factor $\frac{m}{q} log_2(m)$ represents the overhead due to communications and synchronizations. Finally, we have $t_{i,j}(m_i) = t(m_i, j)$ where $t_{i,j}(m_i)$ is the execution time for application T_i with a problem of size m_i on j identical processors.

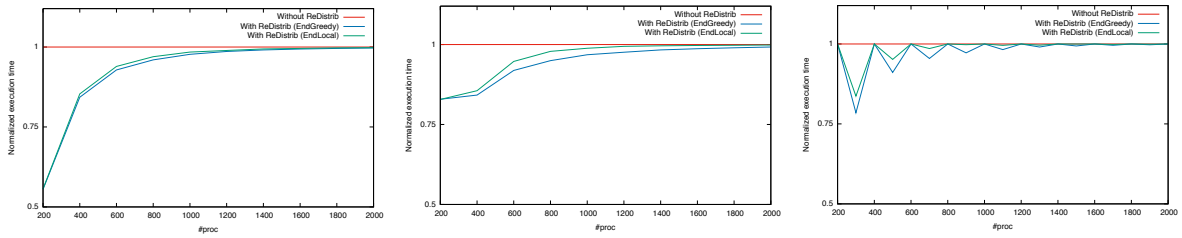
Finally, we assign to each application T_i a random value for the number of data m_i such that: $m_{inf} \leq m_i \leq m_{sup}$. If $m_{inf} \ll m_{sup}$ then the data distribution between applications is very heterogeneous. On the contrary, if m_{inf} is close to m_{sup}, the data distribution is homogeneous, in other words all applications have (almost) the same execution time. Unless stated otherwise, we set $m_{inf} = 1,500,000$ and $m_{sup} = 2,500,000$ to have execution times long enough so that several failures are likely to strike during execution. With such a value for m_{sup}, the longest execution time in a fault-free execution is around 100 days. We also consider two different data distribution cases, (i) very heterogeneous with $m_{inf} = 1,500$, and (ii) homogeneous with $m_{inf} = 2,499,000$.

Algorithm 9: SHORTESTAPPLICATIONSFIRST

procedure SHORTESTAPPLICATIONSFIRST (L, t, f, σ) **begin**
  $\sigma_{init} := \sigma$;
  /* Compute α_i^t */
  for $i = 1$ **to** n **do**
    if $i \neq f$ **then** $\alpha_i^t := \alpha_i - (t - t_{lastR_i} - \left\lfloor (t - t_{lastR_i})/\tau_{i,\sigma(i)} \right\rfloor C_{i,\sigma(i)})/t_{i,\sigma(i)}$;
    else $\alpha_f^t := \alpha_f$;
  end
  $k := p - \sum_{i=1}^{n} \sigma(i)$ /* There are k available processors */;
  if $\sigma(f) + k < j_{max}(f)$ **then**
    $\sigma(f) := \sigma(f) + k$; $improvable := true$;
  else $\sigma(f) := j_{max}(f)$; $improvable := false$;
  $t_f^U := t + RC_f^{\sigma_{init}(f) \to \sigma(f)} + C_{f,\sigma(f)} + t^R_{f,\sigma(f)}(\alpha_f)$;
  /* Taking processors from shortest application */;
  while $improvable$ **do**
    Let T_s be the shortest application such that $\sigma(s) \geq 4$; $improvable := false$; $q := 2$;
    while $q \leq \sigma(s) - 2$ **do**
      $t_f^E := t + RC_f^{\sigma_{init}(f) \to \sigma(f)+q} + C_{f,\sigma(f)+q} + t^R_{f,\sigma(f)+q}(\alpha_f)$;
      $t_s^E := t + RC_s^{\sigma_{init}(s) \to \sigma(s)-q} + C_{s,\sigma(s)-q} + t^R_{s,\sigma(s)-q}(\alpha_s^t)$;
      if $t_f^E < t_f^U$ *and* $t_s^E < t_f^U$ **then** $improvable := true$; $q := \sigma(s) + 1$;
      else $q := q + 2$;
    end
    if $improvable$ **then**
      $\sigma(f) := \sigma(f) + 2$; $\sigma(s) := \sigma(s) - 2$;
      $t_f^U := t + RC_f^{\sigma_{init}(f) \to \sigma(f)} + C_{f,\sigma(f)} + t^R_{f,\sigma(f)}(\alpha_f)$;
      $t_s^U := t + RC_s^{\sigma_{init}(s) \to \sigma(s)} + C_{s,\sigma(s)} + t^R_{s,\sigma(s)}(\alpha_s^t)$;
      if $t_s^U > t_f^U$ **then** $improvable := false$;
    end
  end
  /* Updating α_i and t_{lastR_i} if needed */
  for $i = 1$ **to** n **do**
    if $\sigma(i) \neq \sigma_{init}(i)$ **then**
      $\alpha_i := \alpha_i^t$;
      $t_{lastR_i} := t + RC_i^{\sigma_{init}(i) \to \sigma(i)} + C_{i,\sigma(i)}$;
    end
  end
  return σ;
end

Algorithm 10: ITERATEDGREEDY

procedure ITERATEDGREEDY (L, t, f, σ) **begin**
  $\sigma_{init} := \sigma$;
  for $i = 1$ **to** n **do**
    if $i \neq f$ **then** $\alpha_i^t := \alpha_i - (t - t_{lastR_i} - \left\lfloor (t - t_{lastR_i})/\tau_{i,\sigma(i)} \right\rfloor C_{i,\sigma(i)})/t_{i,\sigma(i)}$;
    $\alpha_f^t := \alpha_f$;
    $\sigma(i) \leftarrow 2$;
    if $\sigma(i) \neq \sigma_{init}(i)$ **then** $t_i^U = t + RC_i^{\sigma_{init} \rightarrow \sigma(i)} + C_{i,\sigma(i)} + t_{i,\sigma(i)}^R(\alpha_i^t)$;
  end
  Let L be the list of applications sorted in non-increasing values of t_i^U;
  $p_{available} := p - 2n$;
  while $p_{available} \geq 2$ **do**
    $T_i := head(L)$; $L := tail(L)$;
    $improvable := false$; $q := 2$;
    while $\sigma(i) + q < j_{max}(i)$ **do**
      if $\sigma(i) + q = \sigma_{init}(i)$ **then** $t^E := t_{lastR_i} + t_{i,\sigma(i)+q}^R(\alpha_i)$;
      else $t^E := t + t_{i,\sigma(i)+q}^R(\alpha_i^t) + RC_i^{\sigma_{init}(i) \rightarrow \sigma(i)+q} + C_{i,\sigma(i)+q}$;
      if $t^E < t_i^U$ **then** $improvable := true$; $q := j_{max}(i)$;
      else $q := q + 2$;
    end
    if $improvable$ **then**
      $\sigma(i) := \sigma(i) + 2$;
      if $\sigma(i) = \sigma_{init}(i)$ **then** $t_i^U := t_{lastR_i} + t_{i,\sigma(i)}^R(\alpha_i)$;
      else
        $t_i^U := t + RC_i^{\sigma_{init} \rightarrow \sigma(i)} + C_{i,\sigma(i)} + t_{i,\sigma(i)}^R(\alpha_i^t)$;
      end
      $L :=$ Insert T_i in L according to its t_i^U value;
      $p_{available} := p_{available} - 2$;
    end
    else $p_{available} := 0$;
  end
  /* Updating t_{lastR_i} and α_i if needed */
  for $i = 1$ **to** n **do**
    if $\sigma(i) \neq \sigma_{init}(i)$ **then**
      $\alpha_i := \alpha_i^t$;
      $t_{lastR_i} := t + RC_i^{\sigma_{init}(i) \rightarrow \sigma(i)} + C_{i,\sigma(i)}$;
    end
  end
  return σ;
end



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.9: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.

The cost of checkpoints for an application T_i with j processors is $C_{i,j} = C_i/j$, where C_i is proportional to the memory footprint of the application. We have $C_i = m_i \times c$, where c is the time needed to checkpoint one data unit of m_i. The default value is $c = 1$, unless stated otherwise. The synchronisation cost value S is fixed to $S = 0$ for all following experiments. Finally, the MTBF of a single processor is fixed to 100 years, unless stated otherwise.

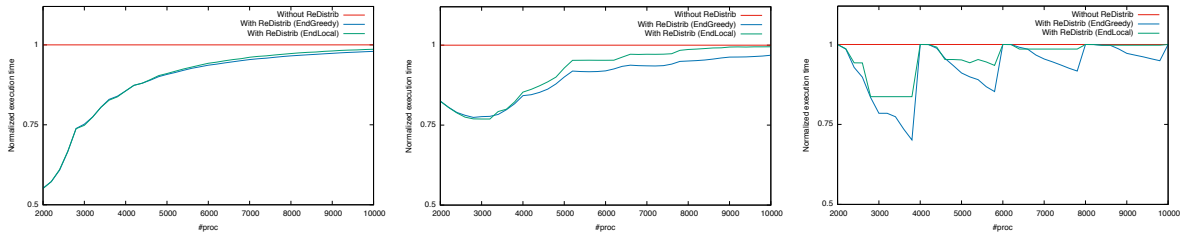
In the following section, we vary the number of processors, the number of applications, the checkpointing cost and the data distribution, in order to study their impact on performance. Note that we assume that a failure can strike during checkpoints but not during downtime, recovery and while the processor is performing some redistribution.

4.5.2 Results

To evaluate the heuristics, we execute each heuristic $x = 50$ times and we compute the average *makespan*, i.e., the longest execution time in the pack. We compare the makespan obtained by the heuristics to the makespan (i) in a faulty context without any redistribution (worst case), and (ii) in a fault-free context with redistributions (best case). We normalize the results by the makespan obtained in a faulty context without any redistribution, which is expected to be the worst case. The execution in a fault-free setting provides us an optimistic value of the execution of the application in the ideal case where no failures occur.

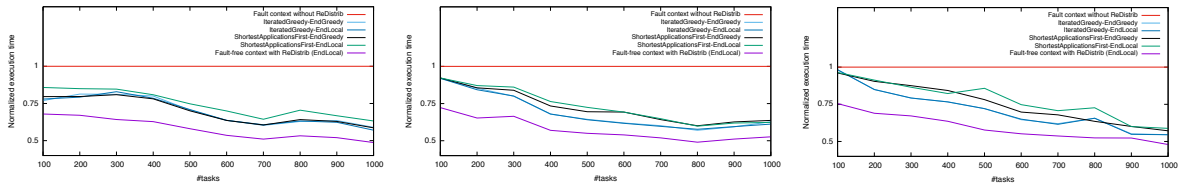
We consider four heuristics: IteratedGreedy-EndGreedy where we greedily recompute a new schedule at each application termination and each failure; IteratedGreedy-EndLocal where we use EndLocal at each application termination, but IteratedGreedy in case of failures; ShortestApplicationsFirst-EndGreedy where we greedily recompute a new schedule at each application termination, but use ShortestApplicationsFirst in case of failures; and ShortestApplicationsFirst-EndLocal where we only use the local variants.

Performance in a fault-free context Figure 4.9 shows the impact of redistribution in a fault-free context with 100 applications, where we vary the number of processors from 200 to 2000. In this case, we compare EndLocal with EndGreedy (see Section 4.4.2). The two heuristics have a very similar behavior, leading to a gain of a least 20% with less than 500 processors, and a slightly better gain for the EndGreedy global heuristic. When the number of processors increases, the efficiency of both heuristics decreases to converge to the performance without redistribution. Indeed, there are then enough processors so that each application does not make use of the extra processors released by ending applications. In the heterogeneous context (with $m_{inf} = 1500$), the gain due to redistribution is even larger.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.10: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

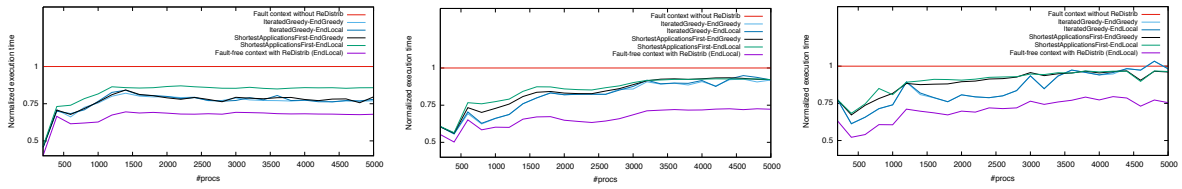
Figure 4.11: Impact of n with $p = 5000$ processors.

Figure 4.10 shows the impact of redistribution in a fault-free context with 1000 applications, we vary the number of processors from 2000 to 10000. We compare ENDLOCAL with ENDGREEDY, the two heuristics have a similar behavior. As showed in Figure 4.9, the redistribution is more efficient in the heterogeneous context (with $m_{inf} = 1500$).

In the homogeneous case (Figure 4.10c), the results clearly show how the number of processors is directly linked to the efficiency of redistribution. If p/n is even, as each application has almost the same size, we assign p/n processors to each application. Consequently each application will finish at the same time and redistributions have very limited use. When p is not divisible by n or if p/n is odd, at least one application will have more processors than the others. So, at least one application will finish before the others and the redistribution will be more efficient. We note that the redistribution has a larger impact when few processors are involved, this is due to the fact that the speedup is better with fewer processors (sublinear speedup). In other words, it is more useful to upgrade from 2 processors to 4 processors rather than from 500 to 502, in terms of speedup.

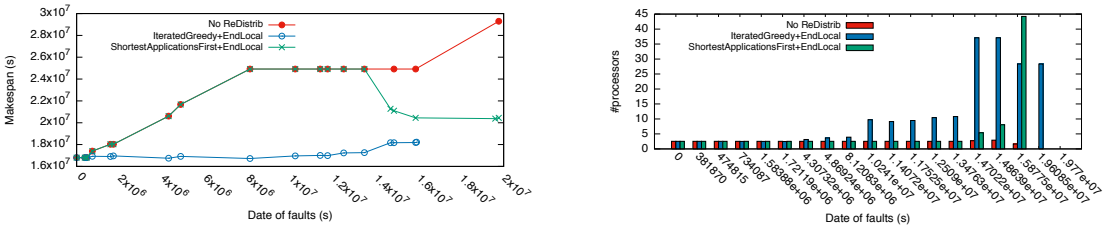
Impact of n Figure 4.11 shows the impact of the number of applications n when the number of processors is fixed to 5000. The results show that having more applications increases the efficiency of both heuristics. With $n = 1000$, we obtain a gain of more than 40% due to redistributions. The reason is that when n increases, the number of processors assigned to each application decreases, then heuristics have more flexibility to redistribute.

Note that, as expected, ITERATEDGREEDY is better than SHORTESTAPPLICATIONSFIRST, because it recomputes a complete new schedule at each fault, instead of just allocating available processors from shortest applications to the faulty application. Using ENDGREEDY with ITERATEDGREEDY does not improve the performance, while ENDGREEDY is useful with SHORTESTAPPLICATIONSFIRST, hence showing that complete redistributions are useful, even when only performed at the end of an application.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.12: Impact of p with $n = 100$ applications and $m_{sup} = 2500000$.



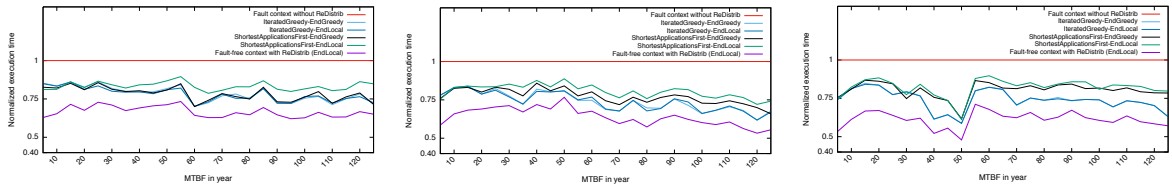
(a) Makespan at each failure dealt with. (b) Standard deviation at each failure dealt with.

Figure 4.13: Heuristic behaviors with $n = 100$, $p = 1000$, MTBF of 50 years, for a single execution.

We also observe that results in the heterogeneous cases are slightly better than in the homogeneous case, but the difference in the homogeneous case when $n = 1000$ is very tiny due to the large number of applications (i.e., fewer processors allocated to applications so the redistribution is more efficient).

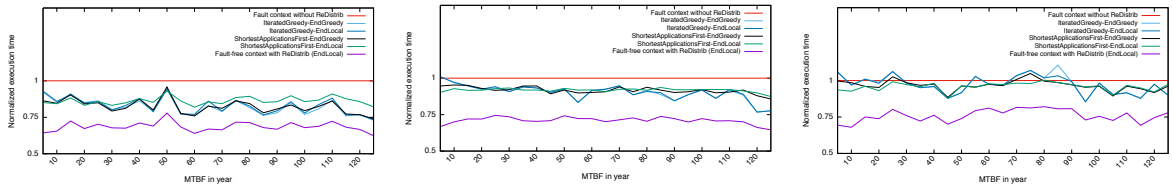
Impact of p Figure 4.12 shows the impact of the number of processors p when the number of applications is fixed. We vary p between 200 and 5000 processors. The results show that having more processors decreases the efficiency of both heuristics, but, in the heterogeneous cases, there is always a gain of at least 10% thanks to redistributions. As noted in the fault-free case, the redistribution is more efficient when the data distribution is very heterogeneous (Figure 4.12a). On the contrary, in the homogeneous case (Figure 4.12c) the redistribution is less efficient (gain around 10%). The same observations hold, i.e., the use of ENDGREEDY vs ENDLOCAL impacts only SHORTESTAPPLICATIONSFIRST. In average, with ITERATEDGREEDY, we obtain a gain of 25%, while SHORTESTAPPLICATIONSFIRST provides a gain around 15% when it is not combined with ENDGREEDY. This figure also allows us to observe the impact of the MTBF on performance. Indeed, the MTBF is set to 100 years for each processor, but the overall MTBF for an application ($\mu_{i,j}$ value) decreases when the number of processors increases, so the gain obtained by the heuristics decreases due to the increasing number of failures.

Heuristic behaviors Figure 4.13 compares ITERATEDGREEDY and SHORTESTAPPLICATIONSFIRST, when combined with ENDLOCAL, on a single execution. We depict both the evolution of the makespan (see Figure 4.13a) and the standard deviation, in terms of number of processors (see Figure 4.13b). ITERATEDGREEDY is clearly superior in terms of makespan, and this can be explained by the fact that it allocates more processors to the longest application, earlier in time than SHORTESTAPPLICATIONSFIRST, hence resulting in a larger standard deviation. Because SHORTESTAPPLICATIONSFIRST takes only local decisions, it takes more time before enough processors are given to the longest application.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.14: Impact of MTBF with $n = 100$, $p = 1000$, and $m_{sup} = 2500000$.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

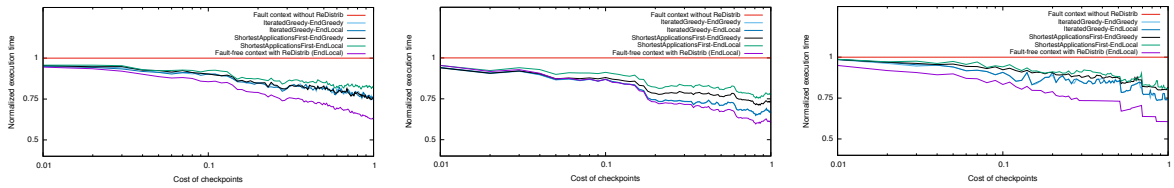
Figure 4.15: Impact of MTBF with $n = 100$, $p = 5000$, and $m_{sup} = 2500000$.

Impact of MTBF Figures 4.14 and 4.15 show the impact of the MTBF on the performance of redistributions. We vary the MTBF of a single processor between 5 years and 125 years. When the MTBF decreases, the number of failures increases, consequently the performance of both heuristics decreases. In Figure 4.14, the performance of ITERATEDGREEDY is closely linked to the MTBF value. Indeed, it tends to favor a heterogeneous distribution of processors (i.e., applications with many processors and applications with few processors). If an application is executed on many processors, its MTBF becomes very small and this application will be hit by more failures, hence it becomes even worse than without redistribution!

We observe the same result in Figure 4.15, especially in the homogeneous case (Figure 4.15c). This effect is even amplified due the number of processors ($p = 5000$) which directly decreases the MTBF and deteriorate the performance (increasing number of faults).

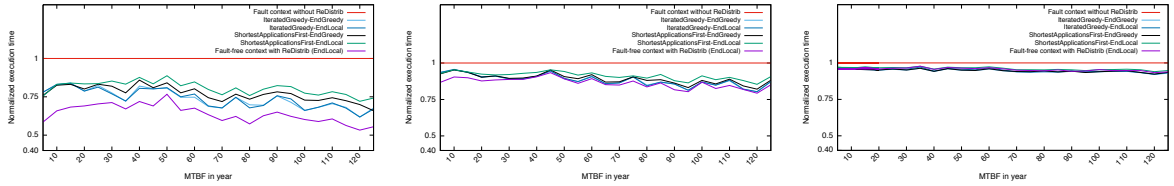
Impact of checkpointing cost Figure 4.16 shows the impact of the checkpointing cost on a platform with 100 applications and 1000 processors. To do so, we multiply the checkpointing cost by c in Figure 4.16 (recall that c is the time needed to checkpoint one data unit). When c decreases, the performance of the heuristics increases and the gap between the execution time in a fault-free context and a fault context becomes small. Indeed, if checkpoints are cheap, a lot of checkpoints can be taken, and the average time lost due to failures decreases. We observe that when the checkpointing cost c tends to 1, the checkpointing costs are more important and the redistribution (specially ITERATEDGREEDY) becomes more unstable. This effect is amplified in a homogeneous context, because applications and checkpoints are larger than in a heterogeneous context. We see the same effect on Figure 4.17.

Impact of the sequential fraction of time Figure 4.18 shows the impact of the sequential fraction of time. We vary f from 0 (applications are fully parallel) to 0.5 (50% of the time is sequential). The results show that when applications are more parallel, the redistribution is more efficient. This result is expected, because if applications are not parallel, there is less gain when trying to allocate more processors to help them complete.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.16: Impact of checkpointing cost.



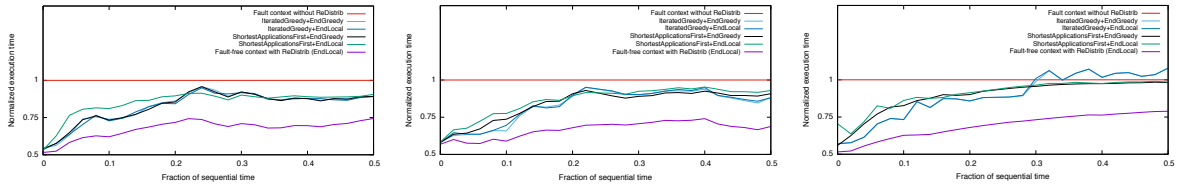
(a) Original checkpoint cost $c = 1$. (b) Checkpoint cost $c = 0.1$. (c) Checkpoint cost $c = 0.01$.

Figure 4.17: Impact of checkpointing cost with $n = 100$, $p = 1000$, and $m_{inf} = 1500000$.

In the homogeneous case (Figure 4.18c), the ITERATEDGREEDY heuristic is worse than the result without redistribution when f is greater than 0.3. It is due to the fact that all applications are large and not fully parallel, so when we greedily recompute a new schedule at each fault, we might deteriorate the performance.

Summary To conclude, we note that ITERATEDGREEDY achieves better performance than SHORTESTAPPLICATIONSFIRST, mainly because it rebuilds a complete schedule at each fault, which is very efficient but also costly. Nevertheless, when the MTBF is low (around 10 years or less), SHORTESTAPPLICATIONSFIRST becomes better than ITERATEDGREEDY. In a faulty context, we gain flexibility from the failures and we can achieve a better load balance. We observe that the ratio between the number of applications and the number of processors plays an important role, because having too many processors for few applications leads to a deterioration of performance (especially in a homogeneous context).

About the data distributions, we observe that the best context to take advantage of redistributions is a heterogeneous context with large and short applications. In the homogeneous context, when we assign the same weight to each application, redistributions become much less interesting. We also show that the cost of checkpointing and the fraction of sequential time have a significant impact on performance.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 4.18: Impact of the sequential fraction of time with $n = 100$ and $p = 1000$ when $0 \leq f \leq 0.5$.

Finally, we point out that all four heuristics run within a few seconds, while the total execution time of the application takes several days, hence even the more costly combination ITERATEDGREEDY-ENDGREEDY incurs a negligible overhead.

4.6 Conclusion

In this chapter, we have designed a detailed and comprehensive model for scheduling a pack of applications on a failure-prone platform, with processor redistributions. We have introduced a greedy polynomial-time algorithm that returns the optimal solution when there are failures but no processor redistribution is allowed. We have shown that the problem of finding a schedule that minimizes the execution time when accounting for redistributions is NP-complete in the strong sense, even with constant redistribution costs and no failures. Finally, we have provided several polynomial-time heuristics to redistribute efficiently processors at each failure or when an application ends its execution and releases processors. The heuristics are tested through extensive simulations, and the results demonstrate their usefulness: a significant improvement of the execution time can be achieved thanks to the redistributions.

Further work will consider partitioning the applications into several consecutive packs (rather than one) and conduct further simulations in this context. We also plan to investigate the complexity of the online redistribution algorithms in terms of competitiveness. It would also be interesting to deal not only with fail-stop errors, but also with silent errors. This would require adding verification mechanisms to detect such errors.

Chapter 5

A performance model to execute workflows on high-bandwidth-memory architectures

Recently, many TOP500 supercomputers [44] use many-core architectures to increase their processing capabilities, such as the Intel Knights Landing (KNL) [61] or some custom many-core architectures [31, 35]. Among these many-core architectures, some systems add a new level in the memory hierarchy: a byte-addressable, high-bandwidth, on-package memory. One of the first widely available systems to exhibit this kind of new memory is the KNL [6, 61, 118]. Its on-package memory (called multi-channel dynamic random access memory, or MCDRAM) of 16 GB has a bandwidth five times larger than the classic double data rate (DDR) memory. At boot, a user can decide to use this on-package memory in three modes:

- **Cache mode**: In cache mode, MCDRAM is used by the hardware as a large last-level direct-mapped cache. In this configuration, cache misses are expensive; indeed, all data will follow the path DDR $\rightarrow$ MCDRAM $\rightarrow$ L2 caches.

- **Flat mode**: In flat mode, the MCDRAM is manually managed by programmers. It is a new fast addressable space exposed as a NUMA node to the operating system.

- **Hybrid mode**: This mode mixes both previous modes. A configurable ratio of the memory is used in cache mode; the other part is configured in flat mode.

While Intel promotes the cache mode, the flat mode may be more interesting in some cases. The goal of this work is to demonstrate, theoretically and experimentally, that the flat mode can obtain better performance with particular workloads (for instance, bandwidth-bound applications). Unlike GPU and classic out-of-core models, with high-bandwidth-memory systems there is no need to transfer the whole data needed for computations into the on-package memory before execution and then to transfer back the data to the DDR after the computation. An application can start its computations using data residing in both memories at the same time.

In this chapter, we build a detailed performance model accounting for the new dual-memory system and the associated constraints. We focus our study on scientific workflows and provide a detailed analysis of the execution time on such platforms, taking into account transfers from both fast and slow memory and their overlap with computations. The problem can be stated as follows: given (i) an application represented as a directed acyclic graph (DAG), and (ii) a many-core platform with P identical processors sharing two memories, a large slow memory and a small fast memory, how should this DAG be scheduled (which processor should execute which task and in which order) and which memory map-

ping should be used (which data should reside in which memory) in order to minimize the total execution time, or *makespan*.

Main contributions. In this chapter, we build a detailed performance model to analyze the execution of workflows on high-bandwidth systems, and we design several scheduling and mapping strategies. We conduct extensive simulations to assess the impact of these strategies on performance. We also conduct experiments for a simple 1D Gauss-Seidel kernel, which establish the accuracy of the model and further demonstrate the importance of a tuned memory management.

The rest of the chapter is organized as follows. Section 5.1 provides an overview of related work. Section 5.2 formally defines the performance model with all its parameters, as well as the target architecture. Section 5.3 discusses the complexity of a particular problem instance, namely, linear workflows. Mapping and scheduling heuristics are introduced in Section 5.4 and evaluated through simulations in Section 5.5. The experiments with the 1D Gauss-Seidel kernel are reported in Section 5.6. Section 5.7 summarizes our conclusions and provides ideas for future work.

5.1 Related work

Deep memory architectures have become widely available only in the last couple of years, and studies focusing on them are rare. Furthermore, since vendors recommend to make use of them as another level of hardware-managed cache, few works make the case for explicit management of these memories. Among existing works, two major trends can be identified: studies arguing for *data placement* or for *data migration*.

Data placement [116] addresses the issue of distributing data among all available memories only once, usually at allocation time. Several efforts in this direction aim at simplifying the APIs available for placement, similarly to work on general NUMA architectures: memkind [32], the Simplified Interface for Complex Memory [70] and Hexe [90]. These libraries provide applications with intent-based allocation policies, letting users specify *bandwidth-bound* data or *latency-sensitive* data, for example. Other works [102, 119] focus instead on tracing the application behavior to optimize data placement on later runs.

Data migration addresses the issue of moving data dynamically across memories during the execution of the application. Preliminary work [97] on this approach showcased that performance of a simple stencil benchmark can be improved by migration, using a scheme similar to out-of-core algorithms, when the compute-density of the application kernel is high enough to provide compute/migration overlapping. Closer to the focus of this chapter, another study [29] discussed a runtime method to schedule tasks with data dependencies on a deep memory platform. Unfortunately, the scheduling algorithm is limited to scheduling a task only after all its input data has been moved to faster memory. Also, no theoretical analysis of this scheduling heuristic was performed.

We also mention the more general field of heterogeneous computing, usually focusing on CPU-GPU architectures. Until recently, these architectures were limited to separated memories: to schedule a task on a GPU, one had to move all of its data to GPU memory. Task scheduling for such architectures is a more popular research area [1, 7, 8, 49]. Unfortunately, the scheduling heuristics for this framework are poorly applicable to our case because we can schedule tasks without moving data first. More recent GPU architectures support accessing main memory (DDR) from GPU code, for example by using unified memory since CUDA 6 [71, 89]. To the best of our knowledge, however no comprehensive study has addressed memory movement and task scheduling for these new GPUs from a performance-model standpoint.

5.2 Model

This section describes the performance model: architecture in Section 5.2.1, the target application in Section 5.2.2, scheduling constraints in Section 5.2.3, execution time in Section 5.2.4, and optimization objective in Section 5.2.5.

5.2.1 Architecture

We consider a deep-memory many-core architecture with two main memories: a large slow-bandwidth memory, M_s, and a small high-bandwidth memory, M_f. This two-unit memory system models that of the Xeon Phi (KNL) architecture [61, 118].

Let S_s denote the size and β_s the bandwidth of the memory M_s. We express memory size in terms of the number of data blocks that can be stored. A data block is any unit convenient for describing the application (e.g. bytes or words). Accordingly, bandwidths are expressed in data blocks per second. Similarly, let S_f denote the size and β_f the bandwidth of the memory M_f.

Both memories have access to the same P identical processing units, called *processors* in the following. Each processor computes at speed s. Figure 5.1 illustrates this architecture, where the fast MCDRAM corresponds to M_f and the slow DDR memory corresponds to M_s.

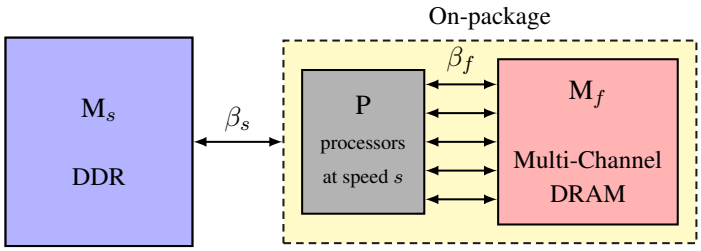


Figure 5.1: Target memory hierarchy.

5.2.2 Application

The target application is a scientific workflow represented by a directed acyclic graph $G = (V, E)$. Nodes in the graph are computation tasks, and edges are dependencies among these computation tasks. Let $V = \{v_1, \ldots, v_n\}$ be the set of tasks. Let $E \subseteq V^2$ be the set of edges. If $(v_i, v_j) \in E$, task v_i must complete its execution before v_j can start. Each task $v_i \in V$ is weighted with the number of computing operations needed, w_i. Each edge $(v_i, v_j) \in E$ is weighted with the number of data blocks shared between tasks, v_i and v_j. Let $e_{i,j}$ be the number of shared (i.e., read or write) data blocks between v_i and v_j. We consider disjoint blocks; hence each $e_{i,j}$ is specific to the task pair (v_i, v_j). For each task, input edges represent data blocks that are read and output edges data blocks that are written. Hence, in the example of Figure 5.2, task v_2 reads $e_{1,2}$ blocks and writes $e_{2,3}$ blocks.

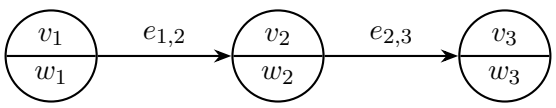


Figure 5.2: Simple DAG example.

We define $\text{succ}(v_i) = \{v_k \mid (v_i, v_k) \in E\}$ (resp. $\text{pred}(v_i) = \{v_k \mid (v_k, v_i) \in E\}$) to be the successors (resp. predecessors) of task $v_i \in V$. Note that if G has multiple entry nodes (i.e., nodes

without any predecessor), then we add a *dummy* node v_0 to G. We set $w_0 = 0$, and v_0 has no predecessor. Finally, v_0 is connected with edges representing the initial input to each entry node of G.

5.2.3 Scheduling constraints

Data blocks. At schedule time, we have to choose from which memory data blocks will be read and written. We define a variable for each edge, $e^f_{i,j}$, which represents the number of data blocks into the fast memory M_f. Symmetrically, let $e^s_{i,j}$ be for each edge the number of data blocks into the slow memory, M_s, defined as $e^s_{i,j} = e_{i,j} - e^f_{i,j}$.

We define $in^f_i = \sum_{v_j \in \text{pred}(v_i)} e^f_{j,i}$ as the total number of blocks read from M_f by task v_i. Similarly, we define $out^f_i = \sum_{v_j \in \text{succ}(v_i)} e^f_{i,j}$ as the total number of blocks written to M_f by task v_i. For the slow memory, M_s, we similarly define in^s_i and out^s_i.

Events To compute the execution time and to express scheduling constraints, we define two events, $\{\sigma_1(i), \sigma_2(i)\}$, for each task v_i. These events are time steps that define the starting time and the ending time for each task. With n tasks, there are at most $2n$ such time steps (this is an upper bound since some events may coincide). A *chunk* is a period of time between two consecutive events. We denote by chunk k the period of time between events t_k and t_{k+1}, with $1 \leq k \leq 2n-1$. Let $t_{\sigma_1(i)}$ be the beginning and $t_{\sigma_2(i)}$ be the end of task v_i (see Figure 5.3). Let $\mathrm{S}^{(k)}_f$ be the number of blocks allocated to the fast memory, M_f, during chunk k. At the beginning, no blocks are allocated; hence we set $\mathrm{S}^{(0)}_f = 0$. At the start of a new chunk k, we first initialize $\mathrm{S}^{(k)}_f = \mathrm{S}^{(k-1)}_f$ and then update this value depending on the events of starting or ending a task. For task v_i, we consider two events (see Figure 5.3):

- At time step $t_{\sigma_1(i)}$: Before v_i begins its execution, the schedule decides which output blocks will be written in fast memory, hence what is the value of $e^f_{i,j}$, for each successor $v_j \in \text{succ}(v_i)$. It must ensure that $\mathrm{S}^{(\sigma_1(i))}_f + out^f_i \leq \mathrm{S}_f$. Thus at time step $t_{\sigma_1(i)}$, out^f_i blocks are reserved in M_f, hence $\mathrm{S}^{(\sigma_1(i))}_f \leftarrow \mathrm{S}^{(\sigma_1(i))}_f + out^f_i$.
- At time step $t_{\sigma_2(i)}$: After computation, we want to evict useless blocks. Since we have disjoint blocks, all read blocks in fast memory are useless after computation; hence $\mathrm{S}^{(\sigma_2(i))}_f \leftarrow \mathrm{S}^{(\sigma_2(i))}_f - in^f_i$. We do not need to transfer these blocks to M_s thanks to the disjoint blocks assumption.

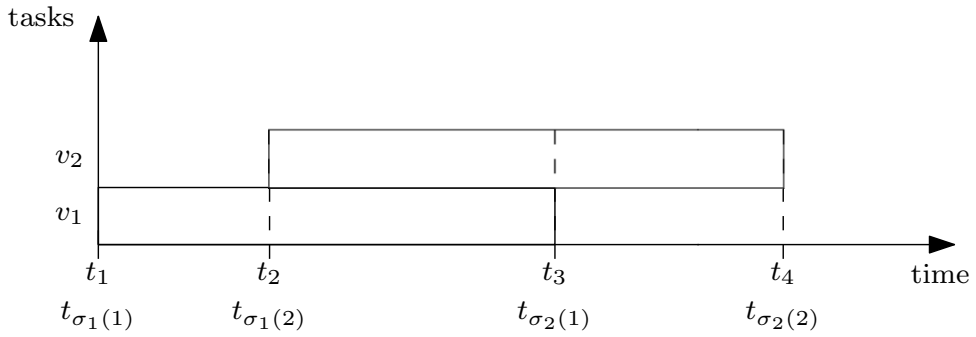


Figure 5.3: Events with two tasks.

To ensure that a task v_i starts only if all its predecessors have finished, we enforce the following constraint:

$$\forall (v_i, v_j) \in E,\ t_{\sigma_2(i)} \leq t_{\sigma_1(j)}. \tag{5.1}$$

Also, we have to ensure that, at any time, the number of blocks allocated in the fast memory, M_f, does not exceed S_f:

$$\forall 1 \leq k \leq 2n-1,\ \mathrm{S}_f^{(k)} \leq \mathrm{S}_f. \tag{5.2}$$

However, we must ensure that no more than P tasks are executing in parallel (no more than one task per processor at any time). Accordingly, we bound the number of executing tasks at each time step t:

$$|\left\{v_i \mid t_{\sigma_1(i)} \leq t < t_{\sigma_2(i)}\right\}| \leq \mathrm{P}. \tag{5.3}$$

We have at most $2n$ events in total, and we have to define a processing order on these events in order to allocate and free memory. We sort the events by nondecreasing date. If two different types of events, $\sigma_1(i)$ and $\sigma_2(j)$, happen simultaneously ($t_{\sigma_1(i)} = t_{\sigma_2(j)}$), then we process $\sigma_2(j)$ first.

5.2.4 Execution time

We aim at deriving a realistic model where communications overlap with computations, which is the case in most state-of-the-art multithreaded environments. We envision a scenario where communications from both memories are uniformly distributed across the whole execution time of each task, meaning that an amount of communication volume from either memory proportional to the execution progress will take place during each chunk, that is, in between two consecutive events, as explained below.

We aim at providing a formula for $w_i^{(k)}$, the number of operations executed by task v_i during chunk k, that is, between time steps t_k and t_{k+1}. If the task v_i does not compute at chunk k, then $w_i^{(k)} = 0$. Otherwise, we have to express three quantities: (i) computations; (ii) communications from and to fast memory, M_f; and (iii) communications from and to slow memory, M_s. We assume that the available bandwidths β_f and β_s are equally partitioned among all tasks currently being executed by the system. Let $\beta_f^{(k)}$ (resp. $\beta_s^{(k)}$) be the available bandwidth during chunk k for memory M_f (resp. M_s) for each task executing during that chunk. Let $N_f^{(k)}$ (resp. $N_s^{(k)}$) be the set of tasks that perform operations using the fast (resp. slow) memory bandwidth. Hence, we have $\beta_f^{(k)} = \frac{\beta_f}{|N_f^{(k)}|}$ and $\beta_s^{(k)} = \frac{\beta_s}{|N_s^{(k)}|}$.

Computations are expressed as the number of operations divided by the speed of the resource used, hence $\frac{w_i^{(k)}}{s}$ for v_i. The task v_i needs to read or write $in_i^f + out_i^f$ blocks in total at speed $\beta_f^{(k)}$. We want to express the communication time between t_k and t_{k+1} also in terms of $w_i^{(k)}$. The number of data accesses in fast memory per computing operations for task v_i can be expressed as $\frac{in_i^f + out_i^f}{w_i}$. The communication time is obtained by multiplying this ratio by the number of operations done during this chunk, $w_i^{(k)}$, and by dividing it by the available bandwidth.

Since each task can perform communications and compute in parallel, we are limited by one bottleneck out of three; computations, or communications from M_f or communications from M_s. Hence, for

each chunk k with $1 \leq k \leq 2n-1$, we have

$$\frac{w_i^{(k)}}{s} \leq t_{k+1} - t_k, \tag{5.4}$$

$$\frac{w_i^{(k)}(in_i^f + out_i^f)}{w_i \beta_f^{(k)}} \leq t_{k+1} - t_k, \tag{5.5}$$

$$\frac{w_i^{(k)}(in_i^s + out_i^s)}{w_i \beta_s^{(k)}} \leq t_{k+1} - t_k. \tag{5.6}$$

Note that a more conservative (and less realistic model) would assume no overlap and replace Equations 5.4 to 5.6 by

$$\frac{w_i^{(k)}}{s} + \frac{w_i^{(k)}(in_i^f + out_i^f)}{w_i \beta_f^{(k)}} + \frac{w_i^{(k)}(in_i^s + out_i^s)}{w_i \beta_s^{(k)}} \leq t_{k+1} - t_k. \tag{5.7}$$

An important assumption is made here: we assume that the number of flops computed with one data block remains constant. In other words, the computation time $\frac{w_i^{(k)}}{s}$ does not depend on the data scheduling (into either fast or slow memory).

From the previous equation, we can derive the expression for $w_i^{(k)}$:

$$w_i^{(k)} = (t_{k+1} - t_k) \min \left(s, \frac{\beta_f^{(k)} w_i}{in_i^f + out_i^f}, \frac{\beta_s^{(k)} w_i}{in_i^s + out_i^s} \right). \tag{5.8}$$

Finally, we need to compute the time step t_{k+1} for the beginning of the next chunk. We express the time $E_i^{(k)}$ for a task i to finish its execution if there are no events after t_k. We call this time the *estimated execution time*, since we do not know whether there will be an event that could modify available bandwidths and change progress rate for the execution of the task:

$$E_i^{(k)} = t_k + \frac{w_i - \sum_{k'=\sigma_1(i)}^{k-1} w_i^{(k')}}{\min \left(s, \frac{\beta_f^{(k)} w_i}{in_i^f + out_i^f}, \frac{\beta_s^{(k)} w_i}{in_i^s + out_i^s} \right)}. \tag{5.9}$$

Hence, the time step of the next event t_{k+1} is

$$t_{k+1} = \min_{v_i \in V} E_i^{(k)}, \tag{5.10}$$

Note that the task that achieves the minimum is not impacted by any event and completes its execution at time step t_{k+1}. We point out that despite the simplifications we made, we still have a complicated model to compute execution time. The reason is that the partitioning of input and output data of each task into fast and slow memory has an impact on the execution of many other tasks, since it imposes constraints on available bandwidth for both memories and remaining space in the fast memory.

There remains to ensure that all tasks perform all their operations and communications. We have the following constraint:

$$\sum_{k=1}^{2n-1} w_i^{(k)} = w_i. \tag{5.11}$$

Indeed, Equation 5.8 guarantees that the communications corresponding to an amount of work $w_i^{(k)}$ can effectively be done during chunk k, since we assume that communications from both memories are uniformly distributed during execution time. Therefore, Equation 5.11 is enough to validate the correctness of computations. Let $in_i^{f(k)} = \frac{w_i^{(k)}}{w_i} in_i^f$ be the number of read operations performed at chunk k by v_i from M_f. We have the following constraint on communications:

$$\sum_{k=1}^{2n-1} in_i^{f(k)} = in_i^f. \tag{5.12}$$

Thanks to Equation 5.11, we ensure that the previous constraint is respected. We have the same type of constraints on in_i^s, out_i^f, and out_i^s. To compute the total execution time of a schedule, we have

$$\mathcal{T} = \max_{v_i \in V} t_{\sigma_2(i)}. \tag{5.13}$$

5.2.5 Objective

Given a directed acyclic graph $G = (V, E)$, our goal is to find a task memory mapping between the small high-bandwidth memory and the large slow-bandwidth memory, in order to minimize the time to execute the critical path of G. More formally, we have the following:

Definition 5.1 (MEMDAG)**.** *Given an acyclic graph $G = (V, E)$ and a platform with P identical processors sharing a two-level memory hierarchy, a large slow-bandwidth memory M_s and a small high-bandwidth memory M_f, find a memory mapping $\mathcal{X} = \{e_{i,j}^f\}_{(v_i,v_j)\in E}$ and a schedule $\{t_{\sigma_1(i)}, t_{\sigma_2(i)}\}_{v_i \in V}$ satisfying all the above constraints and minimizing*

$$\max_{v_i \in V} t_{\sigma_2(i)}.$$

5.3 Complexity for linear chains

MEMDAG is NP-complete in the strong sense. To show this, we remove the memory size constraints and assume an unlimited fast memory with infinite bandwidth. We now have the classical scheduling problem with $n = 3P$ independent tasks to be mapped on P processors, which is equivalent to the 3-partition problem [48]. Since the problem is NP-hard for independent tasks, deriving complexity results for special classes of dependence graphs seems out of reach.

Still, we have partial results for workflows whose graph is a linear chain, as detailed hereafter. Consider a linear chain of tasks

$$v_1 \overset{e_{1,2}}{\to} v_2 \to \cdots \to v_i \overset{e_{i,i+1}}{\to} v_{i+1} \to \cdots \to v_n,$$

and let $e_{0,1}$ denote the input size and $e_{n,n+1}$ the output size. Because of the dependences, each task executes in sequence. Partitioning $e_{i,i+1} = e^s_{i,i+1} + e^f_{i,i+1}$ into slow and fast memory, we aim at minimizing the makespan as follows:

$$\text{MINIMIZE } \sum_{i=1}^{n} m_i$$

$$\text{SUBJECT TO } \begin{cases} e_{i,i+1} = e^s_{i,i+1} + e^f_{i,i+1} & \text{for } 0 \leq i \leq n \\ \frac{w_i}{s} \leq m_i & \text{for } 1 \leq i \leq n \\ \frac{e^s_{i-1,i} + e^s_{i,i+1}}{\beta_s} \leq m_i & \text{for } 1 \leq i \leq n \\ \frac{e^f_{i-1,i} + e^f_{i,i+1}}{\beta_f} \leq m_i & \text{for } 1 \leq i \leq n \\ e^f_{i-1,i} + e^f_{i,i+1} \leq \mathrm{S}_f & \text{for } 1 \leq i \leq n \end{cases} \tag{5.14}$$

Equation 5.14 captures all the constraints for the problem. There are $3n+2$ unknowns, the n values m_i and the $2n+2$ values $e^s_{i,i+1}$ and $e^f_{i,i+1}$. Of course, we can replace one of the latter values, say $e^s_{i,i+1}$, by $e_{i,i+1} - e^f_{i,i+1}$, so there are only $2n+1$ unknowns, but the linear program reads better in the above form.

To solve Equation 5.14, we look for integer values, so we have an integer linear program (ILP). We attempted to design several greedy algorithms to solve Equation 5.14 but failed to come up with a polynomial-time algorithm for an exact solution. We also point out that it is not difficult to derive a pseudo-polynomial dynamic programming algorithm to solve Equation 5.14, using the size S_f of the fast memory as a parameter of the algorithm. Furthermore, on the practical side, we can solve Equation 5.14 as a linear program with rational unknowns and round up the solution to derive a feasible schedule.

Still, the complexity of the problem for linear workflows remains open. At the least, this negative outcome for a simple problem instance, fully evidences the complexity of MEMDAG.

5.4 Heuristics

Since MEMDAG is NP-complete, we derive polynomial-time heuristics to tackle this challenging problem. We have two types of heuristics: (i) processor allocation heuristics that compute a schedule $\mathcal{S}$, defined as a mapping and ordering on the tasks onto the processors and (ii) memory mapping heuristics that compute a memory mapping $\mathcal{X} = \{e^f_{i,j} \,|\, (v_i, v_j) \in E\}$. Recall that when a task finishes its execution, the memory used is released. Therefore, memory mapping is strongly affected by the scheduling decisions. We aim to design heuristics that consider both aspects and minimize the global makespan $\mathcal{T}$.

In Section 5.4.1, we introduce the general algorithm that computes the makespan according to scheduling and memory-mapping policies. Then we present scheduling policies in Section 5.4.2 and memory-mapping policies in Section 5.4.3.

5.4.1 Makespan heuristics

We outline the algorithm to compute the makespan of a task graph according to (i) a processor-scheduling policy called φ and (ii) a memory mapping policy called τ. Let $L^{(k)}$ be the list of ready tasks at time step k. A task is called *ready* when all its predecessors have completed their execution. The scheduling policy, φ, sorts the list of tasks $L^{(k)}$ according to its priority criterion, so that the task in first position in $L^{(k)}$ will be scheduled first. The memory-mapping policy, τ, returns the number of blocks in fast memory for each successor of a task, according to the size of the fast memory available

for this chunk, namely, $S_f - S_f^{(k)}$. In other words, $\tau(v_i)$ returns all $e_{i,j}^f$ with $v_j \in \text{succ}(v_i)$. Algorithm 11 computes the makespan of a task graph G, given a number of processors P, a fast memory of size S_f, and two policies: φ for processors and τ for the memory. The scheduling algorithm is based on a modified version of the *list scheduling* algorithm [81]. The idea of *list scheduling* is to build, at each time step k, an ordered list $L^{(k)}$ of tasks that are ready to be executed. Then, the algorithm greedily chooses the first task in the list if one resource is available at this time step, and so on. The key of list scheduling algorithms lies in the sorting function used to keep the ordered list $L^{(k)}$. We detail several variants in Section 5.4.2. Since we have homogeneous computing resources, we do not need to define a function that sorts computing resources, in order to use the most appropriate one. We simply choose any computing resource available at time-step k.

We now detail the core of the algorithm. At Line 7, we iterate until the list of tasks to execute is empty, in other words until the workflow G has been completely executed. At Line 13, we sort the list of ready tasks at time-step k according to the scheduling policy. At Line 9, we release processors for each task ending at chunk k. At Line 14, we try to schedule all available tasks at time step k, and at Line 17 we choose the memory allocation for each task scheduled. At Line 22, we compute the set of tasks finishing at $k+1$; recall that $E_i^{(k)}$ computes the estimated finishing time of task v_i at chunk k (see Equation 5.10). At Line 25, we compute the list of tasks ready to execute at time step $k+1$.

5.4.2 Scheduling policies φ

The function $\varphi(L^{(k)})$ aims at sorting the list $L^{(k)}$ that contains the ready tasks at step k, in order to decide which tasks should be scheduled first. We define several scheduling policies to schedule tasks onto processors.

Critical path The first heuristic, called critical path (CP), is derived from the well-known algorithm heterogeneous earliest finish time (HEFT) [115]. The HEFT algorithm chooses the task with the highest critical path at each step and schedules this task to a processor that minimizes its earliest finish time. In our model, we consider homogeneous processors; hence we select the first available processor. We define the critical path CP_i of task v_i as the maximum time to execute, without fast memory, any chain of tasks between v_i and an exit task. Formally,

$$CP_i = \max\left(\frac{w_i}{s}, \frac{in_i + out_i}{\beta_s}\right) + \max_{j \in \text{succ}(v_i)} CP_j. \tag{5.15}$$

CP sorts the list of ready tasks according to their critical paths (in nonincreasing order of CP_i).

Gain graph With this heuristic, we avoid short-term decisions that could lead to bad scheduling choices, we take into consideration the potential gain of using fast memory. To estimate the potential gain of a node v_i, we estimate the potential gain of the subgraph rooted at v_i, called G_i.

Definition 5.2 (Rooted subgraph)**.** *Let $G_x = (V_x, E_x)$ be the subgraph rooted at v_x, with $v_x \in V$. The set of vertices $V_x \subseteq V$ contains all nodes in V reachable from v_x. An edge is in $E_x \subseteq E$ if and only if both of its endpoints are in V_x. Formally,*

$$(v_i, v_j) \in E_x \Leftrightarrow v_i \in V_x \text{ and } v_j \in V_x.$$

The gain of using fast memory for a graph is defined as

$$gain(G_i) = \frac{Bl_f(G_i)}{Bl_s(G_i)}, \tag{5.16}$$

Algorithm 11: Compute the makespan of G

procedure MAKESPAN (G, φ, τ, S_f, P) **begin**
  $k \leftarrow 1$;
  $S_f^{(0)} \leftarrow 0$;
  $L^{(k)} \leftarrow \{v_i \text{ s.t } \text{pred}(v_i) = 0\}$; // Roots of G
  $p \leftarrow P$; // Available processors
  foreach $v_i \in V$ **do** $\sigma_1(i) \leftarrow +\infty$; $\sigma_2(i) \leftarrow +\infty$;
  while $L^{(k)} \neq \emptyset$ **do**
    $S_f^{(k)} \leftarrow S_f^{(k-1)}$;
    foreach $v_i \in V$ **s.t.** $\sigma_2(i) = k$ **do**
      $S_f^{(k)} \leftarrow S_f^{(k)} - in_i^f$; // Release input blocks
      $p \leftarrow p + 1$;
    end
    $L^{(k)} = \varphi(L^{(k)})$; // Sort tasks according scheduling policy
    while $p > 0$ **and** $L^{(k)} \neq \emptyset$ **do**
      $v_i \leftarrow \text{head}(L^{(k)})$;
      $L^{(k)} \leftarrow \text{tail}(L^{(k)})$;
      $\{e_{i,j}^f \mid j \in \text{succ}(v_i)\} \leftarrow \tau(v_i)$; // Allocate each $e_{i,j}^f$
      $S_f^{(k)} \leftarrow S_f^{(k)} + out_i^f$; // Allocate output blocks
      $p \leftarrow p - 1$;
      $\sigma_1(i) \leftarrow k$;
    end
    $i \leftarrow \underset{\sigma_1(j) \leq k < \sigma_2(j)}{\arg\min} E_j^{(k)}$; // Finishing task
    $\sigma_2(i) \leftarrow k + 1$;
    $t_{\sigma_2(i)} \leftarrow E_i^{(k)}$;
    $L^{(k+1)} \leftarrow \{v_i \mid \forall v_j \in \text{pred}(v_i) \text{ s.t. } \sigma_2(j) \leq k + 1 < \sigma_1(i)\}$; // Ready tasks for next time-step
    $k \leftarrow k + 1$;
  end
  return $\max_{v_i \in V} t_{\sigma_2(i)}$;
end

where $Bl_f(G_i)$ is the makespan of G_i with an infinite number of processors and with an infinite fast memory and $Bl_s(G_i)$ is the makespan using only slow memory. If $gain(G_i) = 1$, then G_i is compute bound, and using fast memory might not improve efficiently its execution time. The gain graph (GG) heuristic sorts the list of tasks in nondecreasing order of potential gains using fast memory $gain(G_i)$.

5.4.3 Memory mapping policies τ

In addition to scheduling policies with function φ, we need to compute a memory mapping for tasks ready to be scheduled. Recall that the function $\tau(v_i)$ aims at computing the amount of data in fast memory, $e_{i,j}^f$, for each successor of v_i. We propose three heuristics returning a memory mapping.

MEMCP and MEMGG The idea behind these two heuristics is to greedily give the maximum amount of memory to each successor of the task v_i that is going to be scheduled. The difference lies in the criterion used to order the successors. The MEMCP heuristic uses the critical path to choose which successors to handle first (see Algorithm 12), while MEMGG sorts the list of successors in increasing order of their potential gains using fast memory.

Algorithm 12: Heuristic MEMCP

```
1  procedure MEMCP (v_i) begin
2      Let U be the set of v_i's successors ordered by CP_i ;
3      X ← ∅ ;
4      foreach j ∈ U do
5          e^f_{i,j} ← min( S_f − S^(k)_f , e_{i,j} );
6          X ← X ∪ {e^f_{i,j}} ;
7          S^(k)_f ← S^(k)_f + e^f_{i,j} ;
8      end
9      return X ;
10 end
```

MEMFAIR The previous greedy heuristics MEMCP and MEMGG give as much as possible to the first tasks according to their criterion. The idea of MEMFAIR is to greedily give data blocks in fast memory to the tasks, according to their amount of computations, but accounting for other successors. Recall that $\mathrm{S}_f - \mathrm{S}_f^{(k)}$ is the number of blocks available at chunk k. MEMFAIR spreads blocks from fast memory across the successors of the scheduled tasks: each successor has at most a number of blocks equal to $\mathrm{S}_f - \mathrm{S}_f^{(k)}$ divided by the number of successors. Algorithm 13 details this heuristic.

Algorithm 13: Heuristic MEMFAIR

```
1  procedure MEMFAIR (v_i) begin
2      Let U be the set of v_i's successors ordered by w_i ;
3      X ← ∅ ;
4      foreach j ∈ U do
5          e^f_{i,j} ← min( ⌊ (S_f − S^(k)_f) / |succ(v_i)| ⌋ , e_{i,j} );
6          X ← X ∪ {e^f_{i,j}} ;
7          S^(k)_f ← S^(k)_f + e^f_{i,j} ;
8      end
9      return X ;
10 end
```

By combining two heuristics for processor scheduling and three heuristics for memory mapping, we obtain a total of six heuristics.

5.4.4 Baseline heuristics

For comparison and evaluation purposes, we define three different baseline heuristics for memory mapping.

CP+NoFast and CP+InfFast NoFast considers that no fast memory is available, while InfFast uses a fast memory of infinite size (but still with a finite bandwidth, β_f).

CP+CcMode This baseline heuristic is more complicated. Recall that our target architecture is the Xeon Phi KNL, which proposes two principal modes to manage the fast memory: the cache mode and the flat mode [118]. In the cache mode, the fast memory is managed by the system as a large cache. Our memory-mapping heuristic CcMode aims at imitating the KNL cache mode behavior. In CcMode, we divide the fast memory into P slices, where P is the total number of processors and each processor has access only to its own slice into the fast memory. When a node v_i is scheduled onto a processor, all its output blocks are allocated, if possible, to fast memory. If the slice into fast memory is too small to contain the output blocks of each successor, we consider the successors in nondecreasing index order (v_{j-1} is handled before v_j). CcMode aims at providing a more realistic comparison baseline than does NoFast.

5.5 Simulations

To assess the efficiency of the heuristics defined in Section 5.4, we have conducted extensive simulations. Simulation settings are discussed in Section 5.5.1, and results are presented in Section 5.5.2. The simulator is publicly available at `https://perso.ens-lyon.fr/loic.pottier/archives/simu-deepmemory.zip` so that interested readers can instantiate their preferred scenarios and repeat the same simulations for reproducibility purpose.

5.5.1 Simulation settings

To evaluate the efficiency of the proposed heuristics, we conduct simulations using parameters corresponding to those of the Xeon Phi KNL architecture. Unless stated otherwise, the bandwidth of the slow memory, β_s, is set to 90 GB/s, while the fast memory is considered to be five times faster, at 450 GB/s [118]. The processor speed, s, is set to 1.4 GHz (indeed the processor speed of KNL cores ranges from 1.3 to 1.5 with the Turbo mode activated [61]). The size of the fast memory is set to 16 GB unless stated otherwise, and the slow memory is considered infinitely large.

To instantiate the simulations, we use random directed acyclic graphs from the Standard Tasks Graphs (STG) set [114]. The STG set provides 180 randomly generated DAGs with different sizes ranging from 50 to 5,000 nodes. We select two sizes: 50 and 100 nodes. This leads us to two sets of 180 same-size graphs. For these two sets, we further distinguish between sparse and dense graphs. Recall that the density of a graph $G = (V, E)$ is defined as $\frac{|E|}{|V|(|V|-1)}$; hence the density is 0 for a graph without edges and 1 for a complete graph. We consider two different subsets of each set: (i) the 20 graphs, over the 180 available for each set, that exhibit the lower densities and (ii) the 20 graphs with the higher densities in the set. Note that, for practical reasons, we consider only dense graphs of 50 nodes .

We need to set the number of computing operations, w_i, for each node, v_i, in the DAG and the number of data blocks, $e_{i,j}$ (i.e., number of bytes) on each edge. One of the key metrics in task graph scheduling with multiple memory levels is the computation-to-communication ratio (CCR). In our

framework, for a node v_i and an edge $e_{i,j}$, the CCR is the ratio of the time required to compute w_i operations over the time required to transfer $e_{i,j}$ blocks to slow memory:

$$\text{CCR} = \frac{w_i}{s} / \frac{e_{i,j}}{\beta_s}.$$

We let the CCR vary in our experiments and we instantiate the graphs as follows. For the computing part, we choose w_i uniformly between $w_i^{\min} = 10^4$ and $w_i^{\max} = 10^6$ flops: since the processor speed s is set to 1.4 GHz, the computing part of each node is comprised between 10^{-3} and 10^{-5} seconds. For data transfers, we randomly and uniformly pick $e_{i,j}$ in the interval

$$\left[\frac{w_i^{\min} \times \beta_s}{s \times \text{CCR}}, \frac{w_i^{\max} \times \beta_s}{s \times \text{CCR}} \right].$$

5.5.2 Results

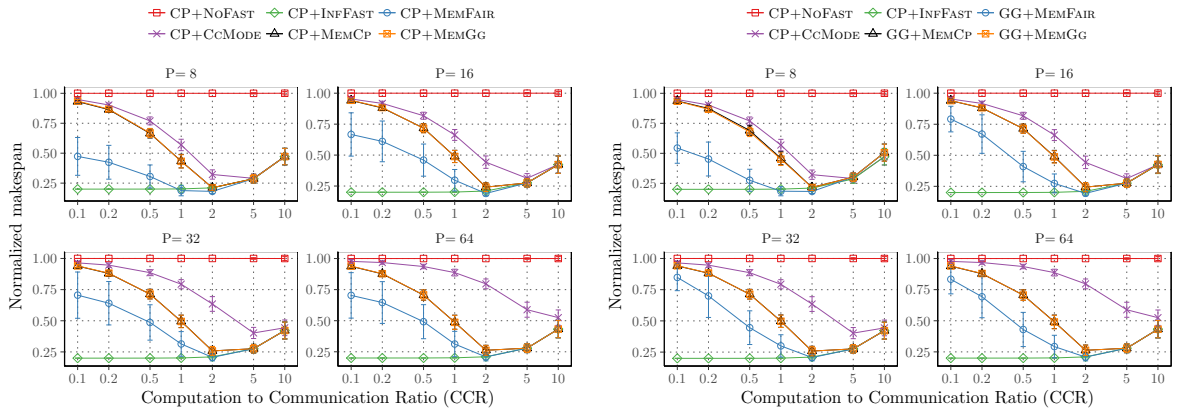
To evaluate the heuristics, we execute each heuristic 50 times with different random weights on the 20 graphs from each STG subset; hence each point is the average of $1,000$ executions. Then, we compute the average makespan over all the runs. All makespans are normalized with the baseline without fast memory, CP+NoFast. The standard deviation is represented as error bars. We study the impact of the number of processors, the size of fast memory, and the fast memory bandwidth, by varying these parameters in the simulations.

Impact of the number of processors

Sparse case. Figure 5.4a presents the normalized makespan of graphs of 50 nodes, and with 1 GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64 with the scheduling policy CP combined with each memory mapping. Figure 5.4b presents the same results but for the scheduling policy GG. All heuristics exhibit good performance in comparison to the two baselines CP+NoFast and CP+CcMode, but only GG +MemFair and CP+MemFair clearly outperform other heuristics, with an average gain around 50% over the baseline CP+NoFast. CP and GG present similar trends; the difference between heuristics performance lies in the memory mapping. With the approaches MemCP and MemGG, we give the maximum number of blocks possible to the successors (according to the heuristic rules). Several nodes might be strongly accelerated but likely at the expense of other nodes in the graph. On the contrary, MemFair aims at giving a fair amount of fast memory to each successor of the scheduled task. As a result, the usage of fast memory is more balanced across tasks in the graph than for mappings produced by MemCP and MemGG.

When the CCR decreases, the number of data blocks on the edges increases, and the graph no longer fits into fast memory. On the contrary, when the CCR increases, the number of data blocks on the edges decreases, so that the graph fits, at some point, into the fast memory, but then computations become the bottleneck, and the benefits of the high-bandwidth memory are less important. For small values of P, MemCP and MemGG show almost the same behavior with noticeable improvements over the case without fast memory NoFast, but are close to the cache mode CcMode. When the number of processors increases, the performance of CcMode decreases, mainly because when P increases, the size of each fast memory slice decreases.

Figure 5.5 presents the normalized makespan of graphs with 100 nodes, and with 1GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64.The results are similar to the case with 50 nodes (see the Figure 5.4), the impact of the size of graphs is not strong, mainly because the performance are strongly linked to the CCR .



(a) CP (b) GG

Figure 5.4: Impact of the number of processors with 50 nodes and $S_f = 1$ GB fast memory for CP and GG scheduling heuristicsfor the sparse case.

Dense case. Figure 5.6a presents the normalized makespan of dense graphs of 50 nodes, and with 1GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64.Compared to the sparse case (see Figure 5.4) all heuristics shows degraded performance, mainly due to the fact that dense graphs are larger than sparse graphs, in terms of memory usage. But, global performance are very good, with an average gain around 50% for the best combination.

Impact of fast memory size

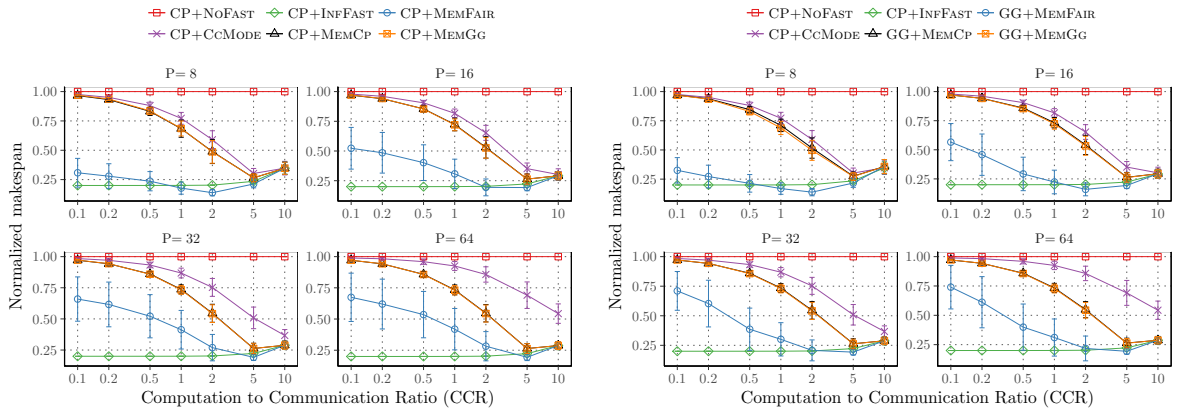
Sparse case. Figure 5.7 presents the results for graphs with 50 nodes, with 8 processors when we vary the fast memory size and the CCR. As always, we vary the CCR from 0.1 to 10 and the size of fast memory from 200MB to 16GB. Recall that, the fast memory bandwidth is set to 450 GB/s (five times faster). Both scheduling heuristics CP and GG show similar performance. Clearly, when the size of the memory is increasing, the global performance of heuristics converges to the baseline CP+InfFast. All proposed heuristics perform better than the cache mode CcMode, and MemFair outperforms other memory mappings with an average gain around 25%, when the size of fast memory is small enough so that all data do not fit in fast memory. We observe that the CCR for which all heuristics reach the lower baseline InfFast decreases when the fast memory size increases.

Figure 5.8 presents the results for graphs with 100 nodes, with 8 processors when we vary the fast memory size and the CCR. The results with 100 nodes are similar to the results with 50 nodes, the memory mapping MemFair performs better with 100 nodes.

Dense case. Figure 5.9 presents the results for dense graphs with 50 nodes, with 8 processors when we vary the fast memory size and the CCR. The results between dense and sparse case show similar trends. The memory-mapping heuristic is more important with dense graphs, mainly because a limited part of a dense graphs can fit in the fast memory; hence the mapping has a strong impact on performance.

Impact of fast memory bandwidth

Sparse case. Figure 5.10 presents the results for graphs with 50 nodes, with 8 processors and 1GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow mem-



(a) CP (b) GG

Figure 5.5: Impact of the number of processors with 100 nodes and $S_f = 1$ GB fast memory for CP and GG scheduling heuristicsfor the sparse case.

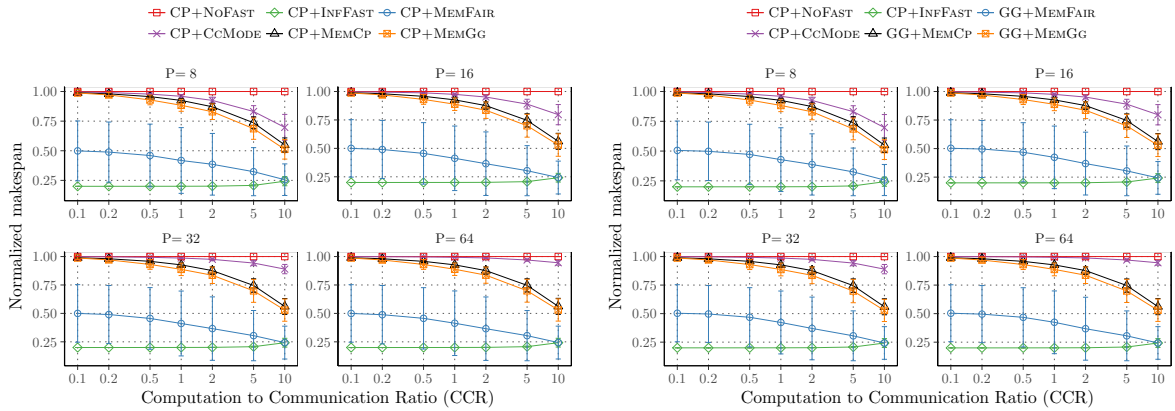
ory bandwidth. Both scheduling heuristics CP and GG exhibit similar performance when we vary the fast memory bandwidth. We observe that for small bandwidths, the memory mapping MEMFAIR outperforms the baseline INFFAST. Recall that the fast memory bandwidth is the same for every memory heuristic, so INFFAST has a infinite fast memory with a finite bandwidth. When the bandwidth is too small compared to the slow memory bandwidth, saturating the fast memory leads to decreased performance because the fast memory bandwidth is shared by the number of tasks concurrently trying to gain access to it.

Figure 5.11 presents the results for graphs with 100 nodes, with 8 processors and 1 GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow memory bandwidth. Results with 100 nodes and with 50 nodes present similar trends, the key point is when the CCR increases the graph no longer fits into the fast memory memory

Dense case. Figure 5.12 presents the results for dense graphs with 50 nodes, with 8 processors and 1 GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow memory bandwidth. We observe similar trends between dense and sparse case, when the CCR increases the performance of all heuristics increase as well. The combinations with MEMFAIR perform the best, with an average gain around 50%.

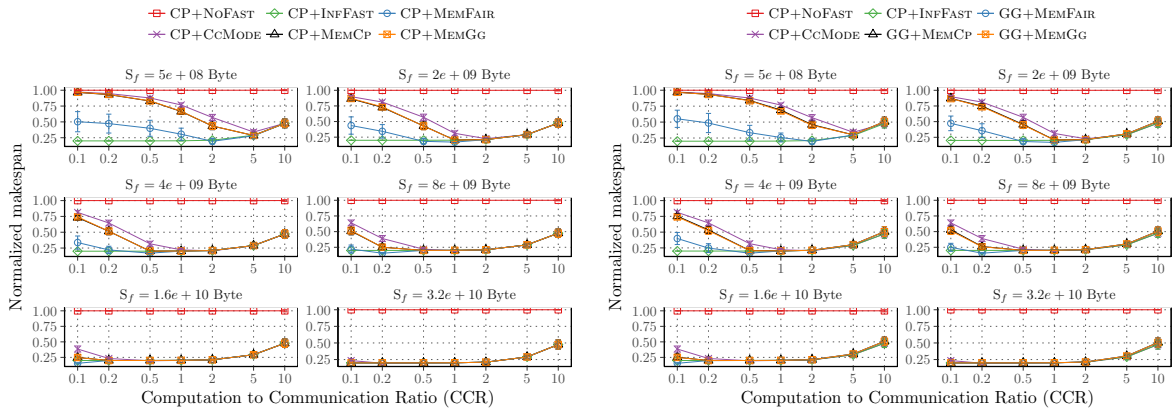
Summary

All heuristics are efficient compared with the baseline without fast memory. But only two combinations, CP+MEMFAIR and GG +MEMFAIR, clearly outperform the baseline CP+CCMODE. Recall that CCMODE aims at imitating KNL's behavior when the system manages the fast memory as a cache. Therefore, obtaining better performance than this mode demonstrates the importance of a fine-tuned memory management when dealing with deep-memory architectures.



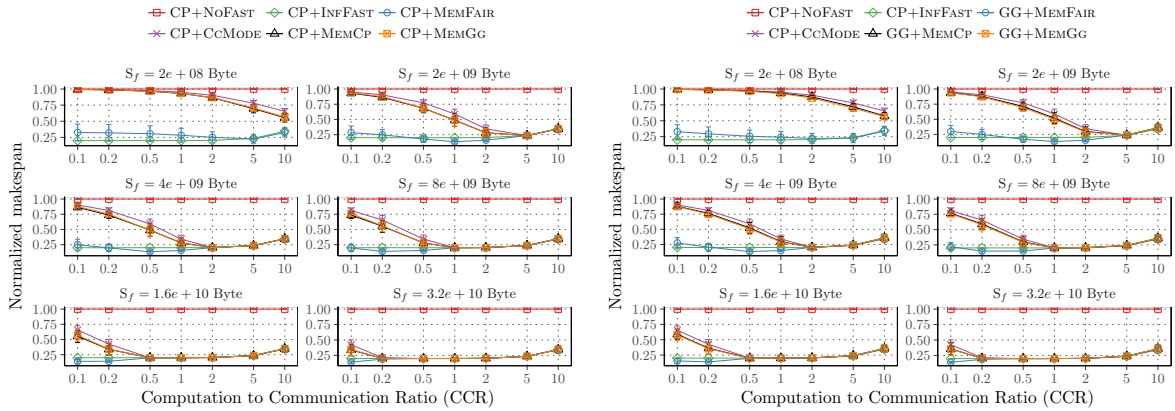
(a) CP (b) GG

Figure 5.6: Impact of the number of processors with 50 nodes and $S_f = 1$GB fast memory for CP and GG scheduling heuristics for dense case.



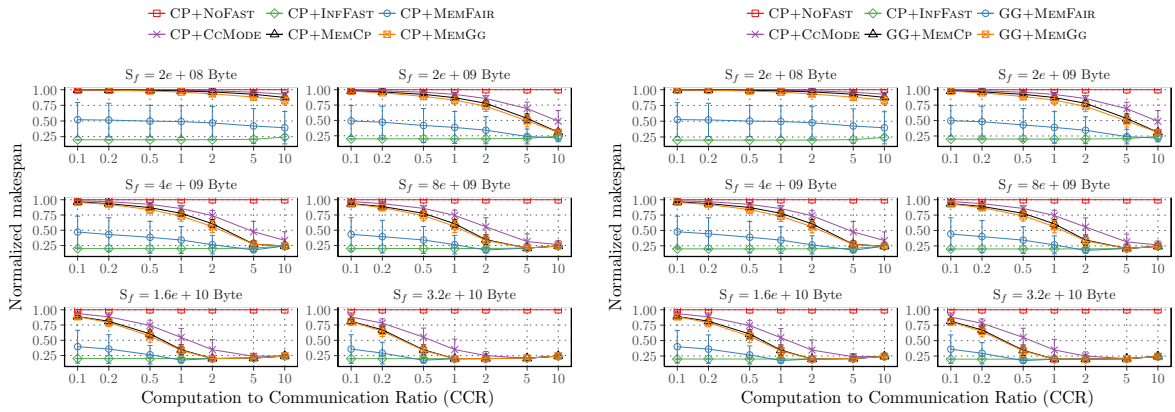
(a) CP (b) GG

Figure 5.7: Impact of fast memory size with 50 nodes and 8 processors for CP and GG scheduling heuristicsfor the sparse case.



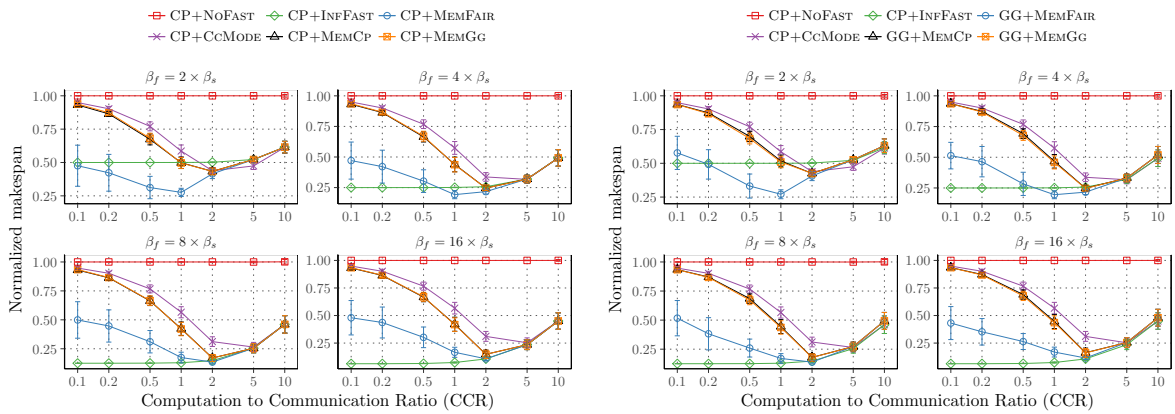
(a) CP (b) GG

Figure 5.8: Impact of fast memory size with 100 nodes and 8 processors for CP and GG scheduling heuristicsfor the sparse case.



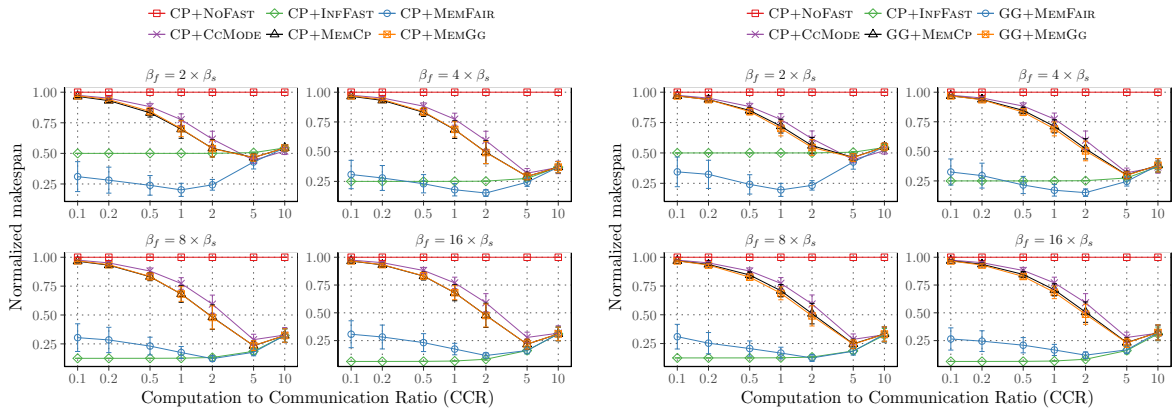
(a) CP (b) GG

Figure 5.9: Impact of fast memory size with 50 nodes and 8 processors for CP and GG scheduling heuristics for the dense case.



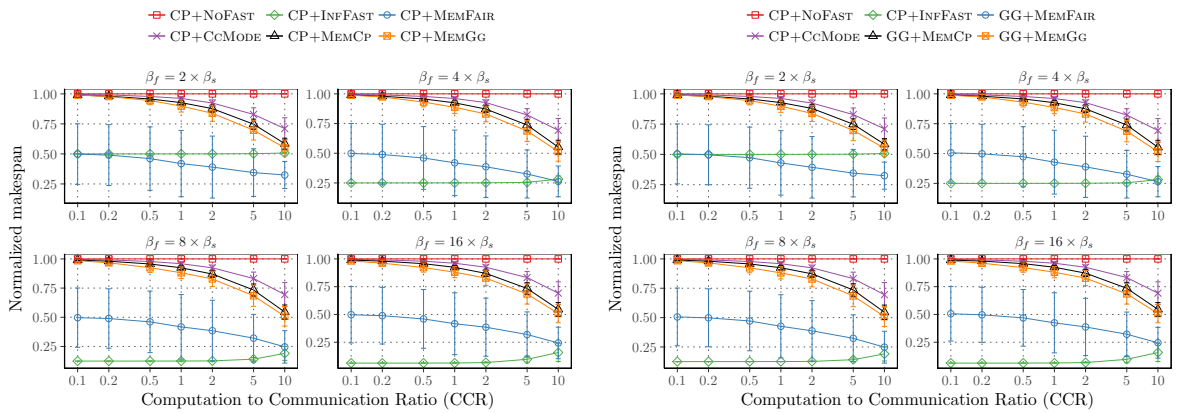
(a) CP (b) GG

Figure 5.10: Impact of fast memory bandwidth with 50 nodes, 8 processors, and $S_f = 1$ GB for CP and GG scheduling heuristicsfor the sparse case.



(a) CP (b) GG

Figure 5.11: Impact of fast memory bandwidth with 100 nodes, 8 processors, and $S_f = 1$ GB for CP and GG scheduling heuristicsfor the sparse case.



(a) CP (b) GG

Figure 5.12: Impact of fast memory bandwidth with 50 nodes, 8 processors, and $S_f = 1$ GB for CP and GG scheduling heuristics for the dense case.

5.6 Experiments

In this section, we assess the accuracy of the model by running both simulations and actual experiments for a 1D Gauss-Seidel computational kernel, using data movement between the slow and fast memories. We detail experimental settings in Section 5.6.1, and present results in Section 5.6.2. The code is available at `https://gitlab.com/perarnau/knl/`.

5.6.1 Experimental settings

Application data is partitioned into rectangular tiles and iteratively updated as shown in Algorithm 14, where Tile_i^t denotes tile i at iteration t.

Algorithm 14: 1D Gauss-Seidel algorithm

```
1 procedure 1D-GS(array) begin
2     for t = 1 to ... do
3         for i = 1 to ... do
4             Tile_i^t ← Gauss-Seidel(Tile_{i-1}^t, Tile_i^{t-1}, Tile_{i+1}^{t-1});
5         end
6     end
7 end
```

At each step of the procedure 1D-GS, Tile_i^t is computed as a combination of three tiles: (i) Tile_{i-1}^t, its left neighbor that has just been updated at iteration t; (ii) Tile_i^{t-1}, its current value from iteration $t-1$; and (iii) Tile_{i+1}^{t-1}, its right neighbor from iteration $t-1$. Each tile is extended with phantom borders whose size depends on the updating mask of the Gauss-Seidel kernel (usually we need one or two columns on each vertical border), so that each tile works on a single file of size m.

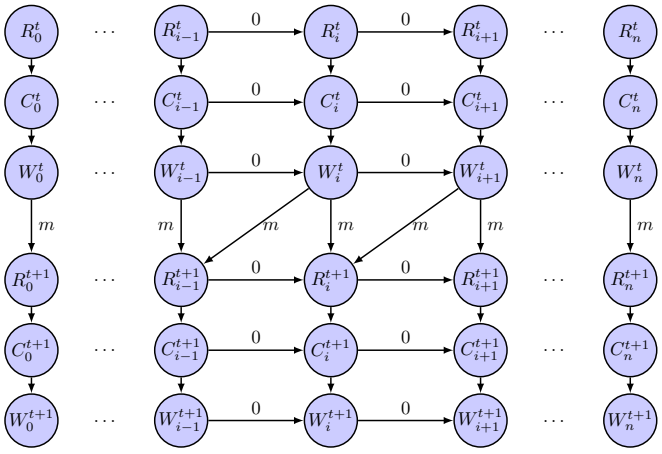


Figure 5.13: 1D stencil task graph, where t is the iteration index, i is the tile index, and m is the size of one tile.

Our model currently does not allow for data movements between the slow and fast memories, so we decompose the update of each tile Tile_i^t into three sequential tasks: (i) task R_i^t transfers the tile from

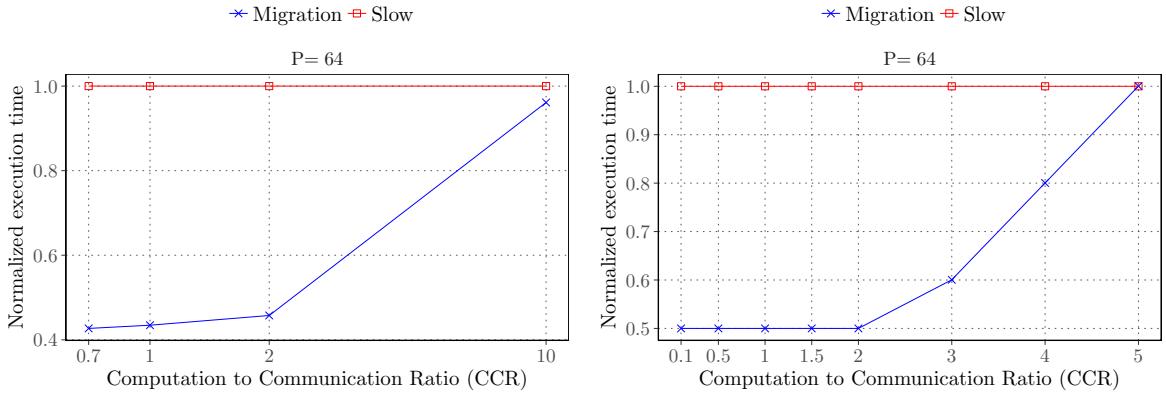
slow memory to fast memory; (ii) task C_i^t computes the tile in fast memory; and (iii) task W_i^t writes the updated tile back into slow memory. This leads to the task graph shown in Figure 5.13. We use this graph as input for the simulations and run the scheduling and mapping heuristics presented in Section 5.4.

For the experiments, we extend the previous study developed for parallel stencil applications in [97] and provide a deep-memory implementation of the 1D Gauss-Seidel kernel for the KNL architecture. First, we copy tiles to migrate input and output data between slow and fast memory. Then, migration tasks and work tasks are pipelined, so that for a given iteration, three batches of tasks are executing concurrently: prefetching of future tiles in fast memory, computing on tiles already prefetched, and flushing of computed tiles back into slow memory. This scheme corresponds to executing tasks R_{i+1}^t, C_i^t and W_{i-1}^t in parallel, as in the classical wavefront execution of the dependence graph in Figure 5.13.

For the experiments, the parameters of the benchmark were the following: (i) input array of 64 GB; (ii) tiles of size 32 MB: (iii) 64 cores at 1.4 GHz; and (iv) 64 threads used. We vary the CCR by increasing the number of operations done per tile.

5.6.2 Results

For the benchmark runs, the platform runs CentOS 7.2, and experiments were repeated 10 times for accuracy. Figure 5.14a gives the performance of the benchmark against a baseline running entirely in slow memory with 64 threads. Figure 5.14b reports the results of the simulations for the same task graph, using the best heuristic, CP+MEMFAIR, on 64 threads.



(a) KNL platform

(b) Model

Figure 5.14: Performance of a 1D stencil with 64 threads.

We observe a good concordance between the experiments and the simulations. In both cases, the performance of the application is greatly increased when using the overlapping scheme and fast memory access. For small values of the CCR, the execution time is divided by half. Then the gain starts to decrease when the CCR reaches the value 2, until reaching a threshold where there is no gain left. This is expected: the threshold is reached when the cost of computations becomes higher than the transfer time of a whole tile from slow memory. We have a discrepancy here since the threshold value is 10 for the experiments and 5 for the simulations. Still, both plots nicely demonstrate the impact of the CCR and the possibility of gaining performance when the CCR is low, hence when access to slow memory is the bottleneck.

5.7 Conclusion

In this chapter, we address the problem of scheduling task graphs onto deep-memory architectures such as the Intel KNL. In addition to the traditional problems of ordering the tasks and mapping them onto processors, a key decision in the scheduling process is what proportion of fast memory should be assigned to each task. We provide a complete and realistic performance model for the execution of workflows on dual-memory systems, as well as several polynomial-time heuristics for both scheduling and memory mapping. These heuristics have been tested through extensive simulations and were shown to outperform the baseline strategies, thereby demonstrating the importance of a good memory-mapping policy. These results also demonstrate that the KNL cache mode can be outperformed by a customized memory mapping. We also conducted experiments on a KNL platform with a 1D Gauss-Seidel computational kernel and compared the performance of a tuned memory mapping with that of the heuristics in simulation, thereby demonstrating the accuracy of the model and bringing another practical proof of the importance of a fine-tuned memory management of the fast memory.

Future work will be devoted to extending simulations on other kinds of workflow graphs, such as fork-join graphs, and extending the model in order to allow for moving data across both memory types. This is a challenging endeavor, because it requires deciding which data blocks to move, and when to move them, while other tasks are executing. Also, we would like to conduct additional experiments with more complicated workflows, such as those arising from dense or sparse linear factorizations in numerical linear algebra. All this future work will rely on the model and results of this chapter, which represent a first, yet crucial, step toward a full understanding of scheduling problems on deep-memory architectures.

Conclusion

In this thesis, we have studied two challenging problems, namely, concurrency and resilience, that must be addressed to cope with future Exascale platforms. In a first time, on the concurrency aspect, we have dealt with the problem of reducing interferences among applications that concurrently use the same last-level cache. Based on a detailed performance model, we have assessed the complexity of the problem and we have designed efficient heuristics. We also have investigated the interest of cache-partitioning techniques on real cache-partitioned multiprocessors platform. In a second time, we have built a model, established the problem complexity and designed efficient heuristics to tackle the problem of co-scheduling applications into a failure-prone context. After focusing on co-scheduling techniques, we have started to investigate the problem of scheduling a workflow on emerging architectures (e.g., many-core) providing a new level of memory. With the advent of the many-core technology in high performance computing, this research topic appears to be quite promising.

Our main contributions in each chapter are summarized in the following paragraphs.

Co-scheduling applications on cache-partitioned systems

In this chapter, we have provided a preliminary work on co-scheduling algorithms for cache-partitioned systems, building upon a theoretical study. The two key scheduling questions are (i) which proportion of cache and (ii) how many processors should be given to each application. For rational numbers of processors, we proved that the problem is NP-complete, but we have been able to characterize optimal solutions for perfectly parallel applications by introducing the concept of *dominant partitions*: for such applications, we have computed the optimal proportion of cache to give to each application in the partition. Furthermore, we have provided explicit formulas to express the number of processors to assign to each application. Several polynomial-time heuristics focusing on Amdahl's applications have been built upon these results, both for rational and integer numbers of processors. Extensive simulation results demonstrate that the use of dominant partitions always leads to better results than more naive approaches, as soon as there is a small sequential fraction of work in application speedup profiles. The concept of sharing the cache only between a subset of applications seems highly relevant, since even an approach with a random selection of applications that share the cache leads to good results.

Co-scheduling HPC workloads on cache-partitioned CMP platforms

Then, from co-scheduling general applications, we have investigated the problem of co-scheduling iterative HPC applications, using the CAT technology provided by Intel to partition the cache. We have proposed a model for the execution time of each application, given a number of cores and a fraction of cache, and we have shown how to instantiate the model on applications coming from the NAS benchmarks. The model turns out to be accurate, as shown in the experiments where we compare the execution time predicted by the model to the real execution time. Several scheduling strategies have been designed, with the goal to maximize the minimum weighted throughput of each application. In particular, we have

introduced an optimal strategy for the model, based upon a dynamic programming algorithm. The results demonstrate that in practice, the optimal strategy often leads to better results than a naive strategy sharing equally the resources between applications. Also, we have determined which combinations of applications benefit most from cache partitioning, and demonstrated the usefulness of cache partitioning.

Resilient co-scheduling of malleable applications

The second main theme of this thesis is resilience. This chapter has addressed the design of a detailed and comprehensive model for scheduling a pack of applications on a failure-prone platform, with processor redistributions. We have introduced a greedy polynomial-time algorithm that returns the optimal solution when there are failures but no processor redistribution is allowed. We have shown that the problem of finding a schedule that minimizes the execution time when accounting for redistributions is NP-complete in the strong sense, even with constant redistribution costs and no failures. Finally, we have provided several polynomial-time heuristics to redistribute efficiently processors at each failure or when an application ends its execution and releases processors. The heuristics are tested through extensive simulations, and the results demonstrate their usefulness: a significant improvement of the execution time can be achieved thanks to the redistributions.

A performance model to execute workflows on high-bandwidth-memory architectures

The last contribution of this thesis is related to the problem of scheduling task graphs onto deep-memory architectures such as the Intel KNL. In addition to the traditional problems of ordering the tasks and mapping them onto processors, a key decision in the scheduling process is what proportion of fast memory should be assigned to each task. We provide a complete and realistic performance model for the execution of workflows on dual-memory systems, as well as several polynomial-time heuristics for both scheduling and memory mapping. These heuristics have been tested through extensive simulations and were shown to outperform the baseline strategies, thereby demonstrating the importance of a good memory-mapping policy. These results also demonstrate that the KNL cache mode can be outperformed by a customized memory mapping. We also conducted experiments on a KNL platform with a 1D Gauss-Seidel computational kernel and compared the performance of a tuned memory mapping with that of the heuristics in simulation, thereby demonstrating the accuracy of the model and bringing another practical proof of the importance of a fine-tuned memory management of the fast memory.

The work conducted in this thesis can be pursued in multiple directions, we discuss here some perspectives.

Perspectives and future work.

Throughout this thesis, at the end of each chapter, we have pointed out several interesting future directions. We present here some hints for further promising research directions.

We have studied the problem of co-scheduling focusing on two aspects, namely, resilience and cache interferences. On the cache side, a short-term perspective is to extend our experimental analysis to other applications and cache-partitioned platforms, to further investigate the potential gains of cache-partitioning on HPC workloads. About long-term perspectives, a first interesting possibility is to extend our analysis to bandwidth-partitioned platforms, a feature recently provided by Intel. In Chapters 2 and 3, we only used cache-partitioning techniques to reduce interferences, but a non negligible part of the interferences occurs in the shared bus between the main memory and the cache. With bandwidth-partitioning, we will be able to strictly restrict the access of both cache and bandwidth to an application

that generates a lot of interferences that slow down other applications. A second perspective is to generalize the experiments to multiprocessors and see if there is a benefit in moving applications from one processor to another, in order to avoid co-locating several cache-intensive applications on the same processor. A third perspective is to find a more suitable law to model caches misses for HPC applications. In Chapters 2 and 3, we used the Power law cache misses to model cache misses behavior. This law gives us an estimation of the number of cache misses given a cache size, but we have showed, experimentally, that this law struggles to model memory-intensive applications. It might be very interesting to validate a new model for cache misses.

On the resilience side, that we explored in Chapter 4, several interesting directions can be considered. The first one is to extend our work to silent-errors, by adding verification mechanisms to detect such errors, and to study the problem with multiple packs instead of one. The second direction is to extend our theoretical analysis to online scheduling problems in a failure-prone context.

Finally, in the last part of this thesis, we initiated a study on the problem of scheduling task-graphs on many-core architectures exhibiting a dual-memory systems. We started by studying classical scheduling approaches, but these many-core architectures often offer a massive concurrency; hence there are well adapted for co-scheduling. Therefore, a very promising research direction would be to apply our co-scheduling model, based on cache partitioning, to these massively parallel dual-memory systems. Indeed we can consider the fast memory as a cache, and use the cache partitioning schemes we have developed on that memory. And, similarly to the bandwidth-partitioned platforms discussed above, we can consider to partition the fast memory and the bandwidth among all concurrent applications, in order to optimize the global platform efficiency.

Bibliography

[1] M. A. Aba, L. Zaourar, and A. Munier. “Approximation Algorithm for Scheduling a Chain of Tasks on Heterogeneous Systems.” In: *European Conference on Parallel Processing*. Springer. 2017, pp. 353–365.

[2] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, et al. “Exascale software study: Software challenges in extreme scale systems.” In: *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009), pp. 1–153.

[3] G. Amdahl. “The validity of the single processor approach to achieving large scale computing capabilities.” In: *AFIPS Conference Proceedings*. 1967, pp. 483–485.

[4] F. Angiolini, L. Benini, and A. Caprara. “Polynomial-time algorithm for on-chip scratchpad memory partitioning.” In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2003, pp. 318–326.

[5] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. “A post-compiler approach to scratchpad mapping of code.” In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2004, pp. 259–267.

[6] R. Asai. *Clustering Modes in Knights Landing Processors: Developer’s Guide*. Tech. rep. Colfax International, May 2016.

[7] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. “Data-Aware Task Scheduling on Multi-accelerator Based Platforms.” In: *IEEE Int. Conf. on Parallel and Distributed Systems*. Dec. 2010, pp. 291–298.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.” In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[9] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan. “Co-scheduling algorithms for high-throughput workload execution.” In: *Journal of Scheduling* 19.6 (2016), pp. 627–640.

[10] O. Avissar, R. Barua, and D. Stewart. “An optimal memory allocation scheme for scratch-pad-based embedded systems.” In: *ACM Transactions on Embedded Computing Systems (TECS)* 1.1 (2002), pp. 6–26.

[11] D. H. Bailey et al. “The NAS Parallel Benchmarks - Summary and Preliminary Results.” In: *Proc. of the 1991 ACM/IEEE Conf. on Supercomputing*. Albuquerque, New Mexico, USA, 1991. ISBN: 0-89791-459-7.

[12] S. Bao, Y. Huo, P. Parvathaneni, A. J. Plassard, C. Bermudez, Y. Yao, I. Llyu, A. Gokhale, and B. A. Landman. “A Data Colocation Grid Framework for Big Data Medical Image Processing-Backend Design.” In: *arXiv preprint arXiv:1712.08634* (2017).

[13] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O’Leary, V. Vishwanath, B. Whitlock, et al. “In situ methods, infrastructures, and applications on high performance computing platforms.” In: *Computer Graphics Forum*. Vol. 35. Wiley Online Library. 2016, pp. 577–597.

[14] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna. "Efficient collective communication in distributed heterogeneous systems." In: *Journal of Parallel and Distributed Computing* 63.3 (2003), pp. 251–263.

[15] R. Bitirgen, E. Ipek, and J. F. Martinez. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach." In: *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. IEEE. 2008, pp. 318–329.

[16] S. Blagodurov, S. Zhuravlev, and A. Fedorova. "Contention-Aware Scheduling on Multicore Systems." In: *ACM Trans. Comput. Syst.* 28.4 (2010), 8:1–8:45.

[17] J. Blazewicz, M. Drabowski, and J. Weglarz. "Scheduling Multiprocessor Tasks to Minimize Schedule Length." In: *Computers, IEEE Transactions on* C-35.5 (May 1986), pp. 389–393. ISSN: 0018-9340.

[18] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram. "Approximation Algorithms for Scheduling Independent Malleable Tasks." English. In: *Euro-Par 2001 Parallel Processing*. Ed. by R. Sakellariou, J. Gurd, L. Freeman, and J. Keane. Vol. 2150. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 191–197. ISBN: 978-3-540-42495-6.

[19] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*. North Holland, 1976.

[20] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. "Unified model for assessing checkpointing protocols at extreme-scale." In: *Concurrency and Computation: Practice and Experience* 26.17 (2014), pp. 2772–2791. ISSN: 1532-0634.

[21] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. "PaRSEC: Exploiting heterogeneity for enhancing scalability." In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45.

[22] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. "DAGuE: A generic distributed DAG engine for high performance computing." In: *Parallel Computing* 38.1-2 (2012), pp. 37–51.

[23] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. "Checkpointing strategies for parallel jobs." In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. Nov. 2011, pp. 1–11.

[24] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A portable programming interface for performance evaluation on modern processors." In: *The international journal of high performance computing applications* 14.3 (2000), pp. 189–204.

[25] F. Cappello and D. Etiemble. "MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks." In: *SC '00*. Washington, DC, USA: IEEE Computer Society, 2000.

[26] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. "Toward exascale resilience." In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 374–388.

[27] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward exascale resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014), pp. 5–28.

[28] D. Chandra, F. Guo, S. Kim, and Y. Solihin. "Predicting inter-thread cache contention on a chip multi-processor architecture." In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE. 2005, pp. 340–351.

[29] K. Chandrasekar, X. Ni, and L. V. Kalé. “A Memory Heterogeneity-Aware Runtime System for Bandwidth-Sensitive HPC Applications.” In: *IEEE Int. Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA*. 2017, pp. 1293–1300.

[30] H. Cho, B. Egger, J. Lee, and H. Shin. “Dynamic data scratchpad memory management for a memory subsystem with an MMU.” In: *ACM SIGPLAN Notices*. Vol. 42. 7. ACM. 2007, pp. 195–206.

[31] P. Computing. *ZettaScaler-2.0 Configurable Liquid Immersion Cooling System*. 2017.

[32] I. Corporation. *Memkind: A User Extensible Heap Manager*. 2018.

[33] J. T. Daly. “A higher order estimate of the optimum checkpoint interval for restart dumps.” In: *FGCS* 22.3 (2004), pp. 303–312.

[34] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel. “A Methodology for Co-Location Aware Application Performance Modeling in Multicore Computing.” In: *Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015, pp. 434–443.

[35] J. Dongarra. “Report on the Sunway TaihuLight system.” In: *PDF). www. netlib. org. Retrieved June* 20 (2016).

[36] J. Dongarra, T. Hérault, and Y. Robert. “Performance and reliability trade-offs for the double checkpointing algorithm.” In: *International Journal of Networking and Computing* 4.1 (2014), pp. 23–41. ISSN: 2185-2847.

[37] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, et al. “The international exascale software project: a call to cooperative action by the global high-performance community.” In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 309–322.

[38] M. Dreher and B. Raffin. “A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations.” In: *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Chicago, United States: IEEE Computer Science Press, May 2014.

[39] M. Drozdowski. “Scheduling Parallel Tasks – Algorithms and Complexity.” In: *Handbook of Scheduling*. Ed. by J. Leung. Chapman and Hall/CRC, 2004. ISBN: 1584883979.

[40] J. Du and J. Y.-T. Leung. “Complexity of Scheduling Parallel Task Systems.” In: *SIAM Journal on Discrete Mathematics* 2.4 (1989), pp. 473–487.

[41] B. Egger, J. Lee, and H. Shin. “Dynamic scratchpad memory management for code in portable systems with an MMU.” In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.2 (2008), p. 11.

[42] B. Egger, J. Lee, and H. Shin. “Scratchpad memory management for portable systems with a memory management unit.” In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. ACM. 2006, pp. 321–330.

[43] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. “A Survey of Rollback-recovery Protocols in Message-passing Systems.” In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408. ISSN: 0360-0300.

[44] Erich Strohmaier et al. *The TOP500 benchmark*. https://www.top500.org/. 2017.

[45] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 78:1–78:12. ISBN: 978-1-4673-0804-5.

[46] M. Frigo, C. E. Leiserson, and K. H. Randall. "The Implementation of the Cilk-5 Multithreaded Language." In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 212–223. ISBN: 0-89791-987-4.

[47] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. "Scheduling the I/O of HPC applications under congestion." In: *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*. 2015, pp. 1013–1022.

[48] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[49] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures." In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 1299–1308.

[50] J. Gecsei, D. Slutz, and I. Traiger. "Evaluation techniques for storage hierarchies." In: *IBM Systems journal* 9.2 (1970), pp. 78–117.

[51] N. Guan, M. Stigge, W. Yi, and G. Yu. "Cache-aware Scheduling and Analysis for Multicores." In: *Proc. 7th ACM Int. Conf. Embedded Software*. EMSOFT '09. ACM, 2009, pp. 245–254.

[52] N. J. Gunther. *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services*. Springer, 2007.

[53] T. Harris, M. Maas, and V. J. Marathe. "Callisto: co-scheduling parallel runtime systems." In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 24.

[54] A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma. "On the nature of cache miss behavior: Is it $\sqrt{2}$." In: *The Journal of Instruction-Level Parallelism* 10 (2008), pp. 1–22.

[55] L. He, H. Zhu, and S. A. Jarvis. "Developing Graph-Based Co-Scheduling Algorithms on Multicore Computers." In: *IEEE Trans. Parallel Distributed Systems* 27.6 (2016), pp. 1617–1632.

[56] M. T. Heath. "A tale of two laws." In: *Int. J. High Performance Computing Applications* 29.3 (2015), pp. 320–330.

[57] T. Herault and Y. Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing, 2015.

[58] T. Herault and Y. Robert. *Fault-tolerance techniques for high-performance computing*. Springer, 2016.

[59] H. Hulett, T. G. Will, and G. J. Woeginger. "Multigraph realizations of degree sequences: Maximization is easy, minimization is hard." In: *Operations Research Letters* 36.5 (2008), pp. 594–596.

[60] Intel. "Intel 64 and IA-32 Architectures Software Developer's Manual." In: *Part 2* 3B: System Programming Guide (2014).

[61] Intel. *Intel Xeon Phi Processor: Performance Monitoring Reference Manual – Volume 1: Registers*. Tech. rep. Intel, Mar. 2017.

[62] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer. "Adaptive insertion policies for managing shared caches." In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 208–219.

[63] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. "Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors." In: *Proc. 17th Int. Conf. Parallel Architectures Compilation Techniques*. PACT '08. ACM, 2008, pp. 220–229.

[64] O.-H. Kang and D. P. Agrawal. "Scalable scheduling for symmetric multiprocessors (smp)." In: *Journal of parallel and distributed computing* 63.3 (2003), pp. 273–285.

[65] S. Kim, D. Chandra, and Y. Solihin. "Fair cache sharing and partitioning in a chip multiprocessor architecture." In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 2004, pp. 111–122.

[66] S. Kim and J. Browne. "A general approach to mapping of parallel computation upon multiprocessor architectures." In: *International conference on parallel processing*. Vol. 3. 1. 1988, p. 8.

[67] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. "Using OS observations to improve performance in multicore systems." In: *IEEE micro* 28.3 (2008).

[68] A. Krishna, A. Samih, and Y. Solihin. "Data sharing in multi-threaded applications and its impact on chip design." In: *Int. Symp. Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2012, pp. 125–134.

[69] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. "Evaluating STT-RAM as an energy-efficient main memory alternative." In: *IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 256–267.

[70] L. A. N. Laboratory. *Simplified Interface to Complex Memory*. 2017.

[71] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. "An investigation of Unified Memory Access performance in CUDA." In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2014, pp. 1–6.

[72] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. "PEBIL: Efficient static binary instrumentation for Linux." In: *IEEE Int. Symp. on Performance Analysis of Systems Software (ISPASS)*. Mar. 2010, pp. 175–183.

[73] J. Leverich and C. Kozyrakis. "Reconciling high server utilization and sub-millisecond quality-of-service." In: *9th European Conf. on Computer Systems*. 2014.

[74] J. Liedtke, H. Hartig, and M. Hohmuth. "OS-controlled cache predictability for real-time systems." In: *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*. IEEE. 1997, pp. 213–224.

[75] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems." In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE. 2008, pp. 367–378.

[76] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. "Improving resource efficiency at scale with Heracles." In: *ACM Transactions on Computer Systems (TOCS)* 34.2 (2016).

[77] P. Malakar, V. Vishwanath, T. Munson, C. Knight, M. Hereld, S. Leyffer, and M. E. Papka. "Optimal scheduling of in-situ analysis for large-scale scientific simulations." In: *Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC'15*. 2015.

[78] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. "Contention aware execution: online contention detection and response." In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2010, pp. 257–265.

[79] G. Martín, D. E. Singh, M.-C. Marinescu, and J. Carretero. "Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration." In: *Parallel Computing* 46 (2015), pp. 60–77. ISSN: 0167-8191.

[80] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. "Scheduling algorithms for effective thread pairing on hybrid multiprocessors." In: *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE. 2005, 10–pp.

[81] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[82] D. Molka, D. Hackenberg, R. Schone, and W. E. Nagel. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture." In: *Int. Conf. on Parallel Processing (ICPP)*. Sept. 2015, pp. 739–748.

[83] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning." In: *Proc. 44th IEEE/ACM Int. Sym. Microarchitecture*. MICRO-44. ACM, 2011, pp. 374–385.

[84] N. Muthuvelu, I. Chai, E. Chikkannan, and R. Buyya. "Batch Resizing Policies and Techniques for Fine-Grain Grid Tasks: The Nuts and Bolts." In: *J. Information Processing Systems* 7.2 (2011).

[85] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. "Fair queuing memory systems." In: *Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture*. IEEE Computer Society. 2006, pp. 208–222.

[86] K. J. Nesbit, J. Laudon, and J. E. Smith. "Virtual private caches." In: *ACM SIGARCH Computer Architecture News* 35.2 (2007), pp. 57–68.

[87] K. T. Nguyen. *Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family*. Feb. 2016.

[88] X. Ni, E. Meneses, and L. Kale. "Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm." In: *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. Sept. 2012, pp. 364–372.

[89] NVIDIA. *CUDA: Unified Memory Programming*. 2018.

[90] L. Oden and P. Balaji. "Hexe: A Toolkit for Heterogeneous Memory Management." In: *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2017.

[91] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. "The case for a single-chip multiprocessor." In: *ACM Sigplan Notices*. Vol. 31. 9. ACM. 1996, pp. 2–11.

[92] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*. July 2013.

[93] J. K. Ousterhout et al. "Scheduling Techniques for Concurrent Systems." In: *ICDCS*. Vol. 82. 1982, pp. 22–30.

[94] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. "Medusa: an experiment in distributed operating system structure." In: *Communications of the ACM* 23.2 (1980), pp. 92–105.

[95] A. J. Pena and P. Balaji. “Toward the efficient use of multiple explicitly managed memory subsystems.” In: *IEEE Int. Conf. on Cluster Computing (CLUSTER)*. Sept. 2014, pp. 123–131.

[96] S. Perarnau, M. Tchiboukdjian, and G. Huard. “Controlling Cache Utilization of HPC Applications.” In: *International Conference on Supercomputing (ICS)*. 2011.

[97] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman. “Exploring Data Migration for Future Deep-Memory Many-Core Systems.” In: *IEEE Cluster*. 2016.

[98] M. K. Qureshi and Y. N. Patt. “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches.” In: *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE. 2006, pp. 423–432.

[99] Advanced Scientific Computing Advisory Committee (ASCAC). *Ten technical approaches to address the challenges of Exascale computing*. 2014.

[100] D. A. Reed, R. Bajcsy, M. A. Fernandez, J.-M. Griffiths, R. D. Mott, J. Dongarra, C. R. Johnson, A. S. Inouye, W. Miner, M. K. Matzke, et al. *Computational science: Ensuring America’s competitiveness*. Tech. rep. President’s Information Technology Advisory Committee Arlington VA, 2005.

[101] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. “Scaling the bandwidth wall: challenges in and avenues for CMP scaling.” In: *ACM SIGARCH Computer Architecture News* 37.3 (2009), pp. 371–382.

[102] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. “Automating the Application Data Placement in Hybrid Memory Systems.” In: *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8*. 2017, pp. 126–136.

[103] R. Sethi and J. D. Ullman. “The generation of optimal code for arithmetic expressions.” In: *Journal of the ACM (JACM)* 17.4 (1970), pp. 715–728.

[104] C. Sewell et al. “Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach.” In: *Proc. of the Int. Conf. for High Perf. Computing, Networking, Storage and Analysis, SC’15*. 2015.

[105] M. Shantharam, Y. Youn, and P. Raghavan. “Speedup-aware co-schedules for efficient workload management.” In: *Parallel Processing Letters* 23.02 (2013), p. 1340001.

[106] T. Sherwood, B. Calder, and J. Emer. “Reducing cache misses using hardware and software page placement.” In: *Proceedings of the 13th international conference on Supercomputing*. ACM. 1999, pp. 155–164.

[107] A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. “Explorations in symbiosis on two multithreaded architectures.” In: *Workshop on Multi-Threaded Execution, Architecture, and Compilers*. 1999.

[108] A. Snavely and D. M. Tullsen. “Symbiotic jobscheduling for a simultaneous mutlithreading processor.” In: *ACM SIGPLAN Notices* 35.11 (2000), pp. 234–244.

[109] G. E. Suh, L. Rudolph, and S. Devadas. “Effects of memory performance on parallel job scheduling.” In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2001, pp. 116–132.

[110] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. “RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations.” In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 1. ACM. 2009, pp. 121–132.

[111] D. Tam, R. Azimi, L. Soares, and M. Stumm. "Managing shared L2 caches on multicore systems in software." In: *Workshop on the Interaction between Operating Systems and Computer Architecture*. Citeseer. 2007, pp. 26–33.

[112] G. Taylor, P. Davies, and M. Farmwald. "The TLB slice-a low-cost high-speed address translation mechanism." In: *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE. 1990, pp. 355–363.

[113] K. Tian, Y. Jiang, and X. Shen. "A Study on Optimally Co-scheduling Jobs of Different Lengths on Chip Multiprocessors." In: *Proc. 6th ACM Conf. Computing Frontiers*. CF '09. ACM, 2009, pp. 41–50.

[114] T. Tobita and H. Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms." In: *Journal of Scheduling* 5.5 (2002), pp. 379–394.

[115] H. Topcuoglu, S. Hariri, and M.-Y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219.

[116] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. D. (Editors), et al. *Programming Abstractions for Data Locality*. Tech. rep. Lugano, Switzerland, Apr. 2014.

[117] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, et al. "An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS." In: *IEEE International Solid-State Circuits Conference, ISSCC 2007, Digest of Technical Papers, San Francisco, CA, USA*. IEEE. 2007, pp. 98–99.

[118] A. Vladimirov and R. Asai. *MCDRAM as High-Bandwith Memory (HBM) in Knights Landing Processors: Developer's Guide*. Tech. rep. Colfax International, May 2016.

[119] G. Voskuilen, A. F. Rodrigues, and S. D. Hammond. "Analyzing allocation behavior for multi-level memory." In: *Proceedings of the Second International Symposium on Memory Systems, MEMSYS 2016, Alexandria, VA, USA, October 3-6, 2016*. 2016, pp. 204–207.

[120] G. V. Wilson. "The history of the development of parallel computing." In: *URL: http://ei.cs.vt.edu/history/Parallel.html* (1994).

[121] Y. Xie and G. Loh. "Dynamic classification of program memory behaviors in CMPs." In: *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*. 2008.

[122] J. W. Young. "A First Order Approximation to the Optimum Checkpoint Interval." In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531. ISSN: 0001-0782.

[123] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. "Smite: Precise QOS prediction on real-system SMT processors to improve utilization in warehouse scale computers." In: *Proc. of the 47th Int. Symp. on Microarchitecture*. 2014, pp. 406–418.

[124] H. Zhu, L. He, B. Gao, K. Li, J. Sun, H. Chen, and K. Li. "Modelling and Developing Co-scheduling Strategies on Multicore Processors." In: *44th Int. Conf. Parallel Processing (ICPP)*. IEEE Computer Society, 2015, pp. 220–229.

[125] S. Zhuravlev, S. Blagodurov, and A. Fedorova. "Addressing shared resource contention in multicore processors via scheduling." In: *ACM Sigplan Notices* 45.3 (2010), pp. 129–142.

[126] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. "Survey of scheduling techniques for addressing shared resources in multicore processors." In: *ACM Computing Surveys (CSUR)* 45.1 (2012), p. 4.

List of publications[1]

Book Chapters

[B1] G. Aupy, A. Benoit, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam. "Co-scheduling high-performance computing applications." In: *Big Data Management and Processing*. Ed. by K.-C. Li, H. Jiang, and A. Zomaya. Chapman and Hall/CRC Press, 2017. Chap. 5.

Articles in International Refereed Journals

[J1] G. Aupy, A. Benoit, S. Dai, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam. "Co-scheduling Amdahl applications on cache-partitioned systems." In: *International Journal of High Performance Computing and Applications* (2017).

[J2] A. Benoit, L. Pottier, and Y. Robert. "Resilient co-scheduling of malleable applications." In: *International Journal of High Performance Computing and Applications* (2017).

Articles in International Refereed Conferences

[C1] A. Benoit, L. Pottier, and Y. Robert. "Resilient application co-scheduling with processor redistribution." In: *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, USA, August 16-19*. Aug. 2016.

[C2] A. Benoit, S. Perarnau, L. Pottier, and Y. Robert. "A performance model to execute workflows on high-bandwidth-memory architectures." In: *47th International Conference on Parallel Processing, ICPP 2018, Eugene, USA, August 13-16*. Aug. 2018.

[C3] G. Aupy, A. Benoit, B. Goglin, L. Pottier, and Y. Robert. "Co-scheduling HPC workloads on cache-partitioned CMP platforms." In: *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13*. IEEE. Sept. 2018.

Articles in International Refereed Workshops

[W1] G. Aupy, A. Benoit, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam. "Co-scheduling algorithms for cache-partitioned systems." In: *19th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2017*. IEEE Computer Society Press, 2017.

[1] Authors are listed in alphabetical order.

Research Reports

[R1] A. Benoit, L. Pottier, and Y. Robert. *Resilient application co-scheduling with processor redistribution*. Research Report RR-8795. INRIA Grenoble - Rhone-Alpes ; ENS de Lyon, Oct. 2015.

[R2] G. Aupy, A. Benoit, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam. *Co-scheduling algorithms for cache-partitioned systems*. Research Report RR-8965. INRIA Grenoble - Rhone-Alpes ; ENS de Lyon, Nov. 2016, p. 28.

[R3] G. Aupy, A. Benoit, S. Dai, L. Pottier, P. Raghavan, Y. Robert, and M. Shantharam. *Co-scheduling Amdahl applications on cache-partitioned systems*. Research Report RR-9021. INRIA Grenoble - Rhone-Alpes ; ENS de Lyon, Feb. 2017, p. 33.

[R4] A. Benoit, S. Perarnau, L. Pottier, and Y. Robert. *A performance model to execute workflows on high-bandwidth memory architectures*. Research Report RR-9165. ENS Lyon ; Inria Grenoble Rhône-Alpes ; University of Tennessee Knoxville ; Georgia Institute of Technology ; Argonne National Laboratory, Apr. 2018, pp. 1–28.

[R5] G. Aupy, A. Benoit, B. Goglin, L. Pottier, and Y. Robert. *Co-scheduling HPC workloads on cache-partitioned CMP platforms*. Research Report RR-9154. Inria, Feb. 2018.