Task Mapping and Load-balancing for Performance, Memory, Reliability and Energy

Changjiang Gou

Numéro National de Thèse : 2020LYSEN047

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

en cotutelle avec

East China Normal University

École Doctorale N°512 :
École doctorale Informatique et Mathématiques de Lyon

Spécialité : Informatique

Soutenue publiquement le 25/09/2020, par :
Changjiang GOU

Task Mapping and Load-balancing for Performance, Memory, Reliability and Energy

Allocation de tâches et équilibrage de charge pour les performances, la mémoire, la fiabilité et l'énergie

---

Devant le jury composé de :

Olivier Beaumont	Directeur de recherche INRIA, INRIA Bordeaux Sud-Ouest	*Rapporteur*
Jean-Marc Nicod	Professeur, FEMTO-ST, ENSMM, Besançon	*Rapporteur*
Florina Ciorba	Assistant professor, Université de Basel, Suisse	*Examinatrice*
Fanny Dufossé	CR INRIA, Université Grenoble Alpes	*Examinatrice*
Alix Munier	Professeure, Sorbonne Université, Paris	*Examinatrice*
Anne Benoit	Maîtresse de conférences, ENS de Lyon, LIP	*Directrice*
Mingsong Chen	Professeur, East China Normal University, Chine	*Co-tuteur*
Loris Marchal	CR CNRS, LIP	*Co-encadrant*

Abstract

This thesis focuses on multi-objective optimization problems arising when running scientific applications on high performance computing platforms and streaming applications on embedded systems. These optimization problems are all proven to be NP-complete, hence our efforts are mainly on designing efficient heuristics for general cases, and proposing optimal solutions for special cases.

Some scientific applications are commonly modeled as rooted trees. Due to the size of temporary data, processing such a tree may exceed the local memory capacity. A practical solution on a multiprocessor system is to partition the tree into many subtrees, and run each on a processor, which is equipped with a local memory. We studied how to partition the tree into several subtrees such that each subtree fits in local memory and the makespan is minimized, when communication costs between processors are accounted for.

Then, a practical work of tree scheduling arising in parallel sparse matrix solver is examined. The objective is to minimize the factorization time by exhibiting good data locality and load balancing. The proportional mapping technique is a widely used approach to solve this resource-allocation problem. It achieves good data locality by assigning the same processors to large parts of the task tree. However, it may limit load balancing in some cases. Based on proportional mapping, a dynamic scheduling algorithm is proposed. It relaxes the data locality criterion to improve load balancing. The performance of our approach has been validated by extensive experiments with the parallel sparse matrix direct solver PaStiX.

Streaming applications often appear in video and audio domains. They are characterized by a series of operations on streaming data, and a high throughput. Multi-Processor System on Chip (MPSoC) is a multi/many-core embedded system that integrates many specific cores through a high speed interconnect on a single die. Such systems are widely used for multimedia applications. Lots of MPSoCs are batteries-operated. Such a tight energy budget intrinsically calls for an efficient schedule to meet the intensive computation demands. Dynamic Voltage and Frequency Scaling (DVFS) can save energy by decreasing the frequency and voltage at the price of increasing failure rates. Another technique to reduce the energy cost and meet the reliability target consists in running multiple copies of tasks. We first model applications as linear chains and study how to minimize the energy consumption under throughput and reliability constraints, using DVFS and duplication technique on MPSoC platforms.

Then, in a following study, with the same optimization goal, we model streaming applications as series-parallel graphs, which are more complex than simple chains and more realistic. The target platform has a hierarchical communication system with two levels, which is common in embedded systems and high performance computing platforms. The reliability is guaranteed through either running tasks at the maximum speed or triplication of tasks. Several efficient heuristics are proposed to tackle this NP-complete optimization problem.

Résumé

Cette thèse se concentre sur les problèmes d'optimisation multi-objectifs survenant lors de l'exécution d'applications scientifiques sur des plates-formes de calcul haute performance et des applications de streaming sur des systèmes embarqués. Ces problèmes d'optimisation se sont tous avérés NP-complets, c'est pourquoi nos efforts portent principalement sur la conception d'heuristiques efficaces pour des cas généraux et sur la proposition de solutions optimales pour des cas particuliers.

Certaines applications scientifiques sont généralement modélisées comme des arbres enracinés. En raison de la taille des données temporaires, le traitement d'une telle arborescence peut dépasser la capacité de la mémoire locale. Une solution pratique sur un système multiprocesseur consiste à partitionner l'arborescence en plusieurs sous-arbres, et à exécuter chacun d'eux sur un processeur, qui est équipé d'une mémoire locale. Nous avons étudié comment partitionner l'arbre en plusieurs sous-arbres de sorte que chaque sous-arbre tienne dans la mémoire locale et que le makespan soit minimisé, lorsque les coûts de communication entre les processeurs sont pris en compte.

Ensuite, un travail pratique d'ordonnancement d'arbres apparaissant dans un solveur de matrice clairsemée parallèle est examiné. L'objectif est de minimiser le temps de factorisation en présentant une bonne localisation des données et un équilibrage de charge. La technique de cartographie proportionnelle est une approche largement utilisée pour résoudre ce problème d'allocation des ressources. Il réalise une bonne localisation des données en affectant les mêmes processeurs à de grandes parties de l'arborescence des tâches. Cependant, cela peut limiter l'équilibrage de charge dans certains cas. Basé sur une cartographie proportionnelle, un algorithme d'ordonnancement dynamique est proposé. Il assouplit le critère de localisation des données pour améliorer l'équilibrage de charge. La performance de notre approche a été validée par de nombreuses expériences avec le solveur direct à matrice clairsemée parallèle PaStiX.

Les applications de streaming apparaissent souvent dans les domaines vidéo et audio. Ils se caractérisent par une série d'opérations sur le streaming de données et un débit élevé. De tels systèmes sont largement utilisés pour les applications multimédias. De nombreux MPSoC fonctionnent sur piles. Un budget énergétique aussi serré nécessite intrinsèquement un calendrier efficace pour répondre aux demandes de calcul intensives. La mise à l'échelle dynamique de la tension et de la fréquence (DVFS) peut économiser de l'énergie en diminuant la fréquence et la tension au prix d'une augmentation des taux de défaillance. Une autre technique pour réduire le coût énergétique et atteindre l'objectif de fiabilité consiste à exécuter plusieurs copies de tâches. Nous modélisons d'abord les applications sous forme de chaînes linéaires et étudions comment minimiser la consommation d'énergie sous des contraintes de débit et de fiabilité, en utilisant DVFS et la technique de duplication sur les plates-formes MPSoC.

Ensuite, dans une étude suivante, avec le même objectif d'optimisation, nous modélisons les applications de streaming sous forme de graphes série-parallèle, plus complexes que de simples chaînes et plus réalistes. La plate-forme cible dispose d'un système de communication hiérarchique à deux niveaux, ce qui est courant dans les systèmes embarqués et les plates-formes informatiques hautes performances. La fiabilité est garantie par l'exécution des tâches à la vitesse maximale ou par la triplication des tâches. Plusieurs heuristiques efficaces sont proposées pour résoudre ce problème d'optimisation NP-complet.

Acknowledgements

First of all, I would like to express my gratitude to Anne Benoit and Loris Marchal. They are my colleges, supervisors and friends. Their expertise guides me in the scheduling field. Often searching the key words given by them saved me a great amount of time. Working with them is really a great pleasure. Discussion with them really helps me focus on the crucial points in the research work. Just list some highlights that I kept in mind. In the first version of our journal paper, I designed the experiments in a very natural and intuitive way: select a heuristic for the first step, then select a heuristic for the second step, and finally compare their performance. Anne asked me why I choose them and if there is any other way to organise it, if there is a better baseline heuristic. Finally we redo the experiments in a different way and dug more information. It changed my stereotype of the experiments section. It's not that easy and straightforward. Taking more parameters into consideration and conducting it in a different way can give us more insights. Through the work about energy minimization under throughput and reliability constraints, Loris and I deduced the period formula on the whiteboard together. In the validation phase, I always try to make it more precise. Then in the next discussion, Loris had to explain me why we should not make it too complex even though it can be more precise. It's the first time in my life that I had the feeling that the models are developed by us. There are many moments like this, I am impressed by their way of handling the hard problems and making things progress.

Many thanks to Prof. Mingsong Chen, my co-advisor from East China Normal University, for his constant support on many topics, like the research directions, courses to take. His wise suggestions improved greatly several works of this thesis.

Thanks for Mathieu Faverge from INRIA Bordeaux and Grégoire Pichon, the young star of our team. We had a nice co-work about improving load-balancing and data locality. Grégoire gave me lots of useful tips on using PlaFRIM and interpreting experiments results. With his help, I can focus on the scheduling side. At the beginning of this cooperation, I misunderstood the models and hence all algorithms developed didn't meet our expectations. I felt helpless and had no idea what to do. Things get better after I spent one week visiting Mathieu at Bordeaux. It's the extensive discussion that makes the program progress towards a promising direction. I am grateful to Matheiu for this cooperation opportunity that improved my communication skills.

Also, I appreciate the friendly atmosphere in ROMA team. Everyone is very kind. Sometimes, we discuss about the technical issues and look for solutions together. More often, the topics are sports, music festivals, lumière festivals and other things happen at Lyon. I am surprised by Frédéric Vivien. Each time I gave him a topic, he can continue and develop a whole story. Thanks for him, even though I understood only part of what he said, I felt I know more about France. Last but not least, many thanks to Laetitia Lecot, Evelyne Blesle and Marie Bozo. With their help, the administration procedures are quite simple, and everything in the daily life at LIP is really well organised.

This Ph.D program is supported by China Scholarship Council (CSC) and PRoSFER program.

Contents

Chapter 1

Introduction

To understand and solve complex problems, scientific computing is nowadays a critical way for research in many fields, for instance biological, physical and social science. Some experiments in these areas may be too long, too expensive or even too dangerous to run in laboratory, simulations through computer software instead provide an alternative solution. Scientific computing, also known as numerical analysis, concerns about the design and analysis of algorithms for solving mathematical problems that arise in modeling physical processes. With the ever increasing amount of datasets generated by instruments, it demands an intensive computing power to have results with higher resolution and higher accuracy in a reasonably short time. Achieving this goal is not trivial.

Parallel and distributed computing systems have been an essential infrastructure for scientific computing for a long time. Commodity processors are grouped into clusters, then these clusters are connected by a network to form a supercomputer, see examples listed on Top500 [82]. Scientific computing application usually has the form of a graph: each node represents a task, and edges among tasks represent data dependencies. Execution of such scientific applications can be seen as a traversal of nodes. Say the execution starts from the source node. After the execution, output of the source then will be sent to its successors. Each successor can start its own workload after the output file is received, this process continues toward terminal nodes. This graph exhibits which tasks can be executed simultaneously and which data they need to access. Since task's size and data dependencies are explicitly showed, task-based scheduling paradigm has been widely accepted in high performance computing (HPC) domain. For instance, various runtime systems, e.g., StarPU [8], Quark [93], and SuperMatrix [24] are designed based on it. Partitioning the graph and mapping subgraphs onto clusters have the potential to reduce execution time dramatically through executing parallel parts simultaneously. In addition, the power of CPU (Central Processing Units) has been increased twice every 18 months in the past decades, following Moore's law. But memory size, speed of read from/write to different levels of memory, and network bandwidth didn't increase at the same rate. The gap still exists. Hence running out of memory, low IO bandwidth, and high-latency of data transfer between clusters could be bottlenecks for high performance goal. Dedicated scheduling and mapping algorithms are indispensable for running scientific applications on massively parallel computing platforms effectively and efficiently.

Applications mentioned above are usually executed in a mode called batch. Scientists have all the raw data, and expect a final result from the application. Applications in batch model are lengthy but not time sensitive. On the contrary, another sort of applications, named streaming applications, are fast and time-sensitive. Streaming data is continuously generated from scientific programs, for instance, data flow from all four experiments in Large Hadron Collider is anticipated to be 25GB/s [22]. Streaming applications motivate part of this work. Streaming applications are defined as applications that process datasets that arrive at a given rate continuously. As datasets continuously enter, the application has to handle them in (near) real time, otherwise the system fails. A classical example is

self-driving cars where multiple sensors transfer datasets continuously to the central control unit, and it has to be able to judge in a very short time if there is a collision in front of the car or it should accelerate to avoid a congestion, otherwise it may cause an accident. The whole application is kept running with a high throughput and has to be highly reliable.

Streaming applications are also very common in image and video processing, e.g. H264 decoder [85], MRI [52], and CT [91], computing platforms are often mobile devices in these scenarios. It can be represented as task graphs as well. This graph provides the possibility of automatic parallelism as it gives the compiler more room to optimize the program through a finer grain parallelism. It favours a computing platform with many processing elements and a desired power/performance ratio [15].

Multiprocessor System-on-Chips (MPSoC) composed of hundreds of processors cores are adopted widely for streaming applications as it achieves an intensive computing power without overwhelming growth in power consumption. Cores are equipped with their own local memory and are connected tightly by an interconnection fabric in MPSoC, the real-time performance are more likely to be guaranteed as resources conflict among tasks are reduced and elements' behaviors are more predictable [90]. MPSoC in mobile devices are often battery-operated. To fit the tight energy budget and heat control, sophisticated power management, such as Dynamic Voltage and Frequency Scaling (DVFS) is proposed to reduce cores' execution voltage when they are handling light-weight tasks. At the same time, the small size of feature devices and the low operating voltage make processing elements vulnerable to radiation-induced soft-errors. When a soft-error hits a core, the tasks running on it need to be re-executed, which incurs further energy cost and may even violate the throughput goal. Hence, given throughput and reliability constraints, how to achieve a fault-tolerant schedule that minimizes the energy consumption for running streaming applications on MPSoC platform is becoming a major challenge.

This manuscript explores task-based scheduling arising in scientific computing domain and streaming domains. Scientific computing on large-scale parallel computing platforms needs to fully take advantage of resources, hence workload balance, low IO and/or memory consumption are still of a great concern. The first part of this manuscript revisits these classical topics and proposes some promising approaches. Another subject considered is running streaming applications on battery-operated mobile devices. Indeed, various sensors and easy access of internet make streaming applications running on mobile devices prevalent. The goal in designing scheduling algorithms for streaming applications is power/performance ratio instead of performance only. The second part of this manuscript studies task-based scheduling of streaming applications to achieve a high throughput and low energy consumption. The main contributions of each chapter are summarized below.

- Chapter 2: Partitioning tree-shaped task graphs for distributed platforms with limited memory [J1] [C4].
  Under this background, the first work of this manuscript is to schedule a tree-shaped task graph onto a homogenous parallel platform with a limited memory size. The tree-shaped task graph comes from factorization of sparse matrix [63], in which most of the elements are zeros. Sparse matrix arises in dealing with partial differential equations, which are very common in scientific or engineering applications. Due to the irregular size of edges and nodes, a different execution order of tasks may have a huge difference on memory consumption. The traversal with minimum memory consumption has been extensively studied and some polynomial algorithms have been proposed by [61] and [50]. In the case that a minimum memory consumption exceeds local memory capability, a good way to solve this problem is to partition the tree into many connected subtrees and map each subtree onto a processor. Chapter 2 explores how to partition the tree such that each subtree fits into local memory and the total execution time is minimized.

- Chapter 3: Improving mappings for sparse direct solvers: a trade-off between data

locality and load balancing [C2].
In the work mentioned in the previous chapter, we limit a processor to be assigned to only a subtree. If a processor can be assigned to two or more subtrees, then data transfer between them will be saved. In this chapter, we focus on mapping subtrees onto processors with a given partition. The aim is to reduce the execution time and keep a good data-locality instead of an acceptable memory consumption. Tree-shaped task graphs are utilized in parallel sparse direct solvers to express parallelism. One of the pre-processing stages of sparse direct solvers consists of mapping computational resources (processors) to these tasks. The objective is to minimize the factorization time by exhibiting good data locality and load balancing. The proportional mapping technique is a widely used approach to solve this resource-allocation problem. It achieves good data locality by assigning the same processors to large parts of the tree. However, it may limit load balancing in some cases. In this chapter, we propose a dynamic mapping algorithm based on proportional mapping. This new approach, named Steal, relaxes the data locality criterion to improve load balancing. The heuristic we proposed is implemented and validated in the PaStiX sparse direct solver.

- Chapter 4: Reliability-aware energy optimization for throughput-constrained applications on MPSoC [C3].
  This chapter aims at reaching an energy efficient, high throughput and reliability scheduling of streaming applications that have the form of a linear chain on MPSoC. Due to limited energy budget, it is hard to guarantee that applications on MPSoC can be accomplished on time with a required throughput. The situation becomes even worse for applications with high reliability requirements, since extra energy will be inevitably consumed by task re-executions or duplicated tasks. The failure rate of cores is modeled as a function of operating frequency/voltage. Based on Dynamic Voltage and Frequency Scaling (DVFS) and task duplication techniques, this chapter presents a novel energy-efficient scheduling model, which aims at minimizing the overall energy consumption of MPSoC applications under both throughput and reliability constraints. The goal is to decide which tasks to duplicate, and at which frequency to operate each task.

- Chapter 5: Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms [R1, C1].
  In a following work, detailed in Chapter 5, the streaming application is modeled as a series-parallel graph (SPG), which covers a larger extent of streaming applications than linear chains. To limit the complexity of the problem, we relax a little the reliability target. Running at the maximum speed causes very few errors, which is acceptable [64]. Indeed, as shown in Chapter 4, soft-errors not necessarily cause throughput violation since tasks can take advantage of idle time slots and catch up later. Then we focus on scheduling of SPG onto a platform with hierarchical communications so that a high performance and low power budget can be met at the same time. The question we need to answer is which tasks should be mapped onto the same core, and onto which core, if they need to be triplicated on three different cores or running at the maximum speed on one core.

Chapter 2

Partitioning tree-shaped task graphs for distributed platforms with limited memory

This study is based on a previous work [50], which studied the complexity of memory minimizing tree traversal in a two-level memory system, and how to reduce the volume of Input/Output in an out-of-core execution. Based on the exact in-core traversal algorithm proposed in [50], we target a multiprocessor system in which each processor has its own local memory. The goal is to minimize the makespan under the memory constraint. We proved that this problem is NP-complete and proposed some heuristics. Preliminary results have been published at PDP 2018 [C4]. Then we extended it by proposing some new heuristics. Extensive simulations with different settings on real trees arising in the context of sparse matrix solvers show that these new heuristics are more general, intuitive and effective than those proposed before. The extended work has been published at TPDS [J1].

2.1 Introduction

Parallel workloads are often modeled as directed acyclic graphs of tasks. We aim at scheduling some of these graphs, namely rooted tree-shaped workflows, onto a set of homogeneous computing platforms, so as to minimize the makespan. Such tree-shaped workflows arise in several computational domains, such as the factorization of sparse matrices [29], or in computational physics code modeling electronic properties [57]. The vertices (or nodes) of the tree typically represent computation tasks, and the edges between them represent dependencies, in the form of output and input files.

In this chapter, we consider out-trees, where there is a dependency from a node to each of its child nodes (the case of in-trees is similar). For such out-trees, each node (except the root) receives an input file from its parent, and it produces a set of output files (except leaf nodes), each of them being used as an input by a different child node. All its input file, execution data and output files have to be stored in local memory during its execution. The input file is discarded after execution, while output files are kept for the later execution of the children.

The potentially large size of the output files makes it crucial to find a traversal that reduces the memory requirement. In the case where even the minimum memory requirement is larger than the local memory capacity, a good way to solve the problem is to partition the tree and map the parts onto a multiprocessor computing system in which each processor has its own private memory and is responsible for a single part. Partitioning makes it possible to both reduce memory requirement and to improve the processing time (or makespan) by doing some processing in parallel, but it also incurs communication costs. On modern computer architectures, the impact of communications between processors on both time and energy is non negligible, furthermore in sparse solvers it can be the bottleneck at even a

small core counts [76]. The problem of scheduling a tree of tasks on a single processor with minimum memory requirement has been studied before, and memory optimal traversals have been proposed [62, 50]. The problem of scheduling such a tree on a single processor with limited memory is also discussed in [50]: in case of memory shortage, some input files need to be moved to a secondary storage (such as a disk), which is larger but slower, and temporarily discarded from the main memory. These files will be retrieved later, when the corresponding node is scheduled. The total volume of data written to (and read from) the secondary storage is called the Input/Output volume (or I/O volume), and the objective is then to find a traversal with minimum I/O volume (MinIO problem).

In this work, the platform is homogeneous, as all processors have the same computing power and the same amount of memory. In the case of memory shortage, rather than performing I/O operations, we send some files to another processor that will handle the processing of the corresponding subtree. If the tree is a linear chain, this will only slow down the computation since communications need to be paid. However, if the tree is a fork graph, it may end up in processing different subtrees in parallel, and hence potentially reducing the makespan. We propose to partition the tree into parts that are connected components, such that each part corresponds to a tree (which is embedded in the whole task tree). The time needed to execute such a part is the sum of the time for the communication of the input file of its root and the computation time of each task in the part. The MinMakespan problem then consists in dividing the tree into parts that are connected components, each part being processed by a separate processor, so that the makespan is minimized. The memory constraint states that we must be able to process each part within the limited memory of a single processor.

The main contributions of this chapter are the following:

- We formalize the MinMakespan problem, and in particular we explain how to express the makespan given a decomposition of the tree into connected components;
- We prove that MinMakespan is NP-complete;
- We design several polynomial-time heuristics aiming at obtaining efficient solutions;
- We evaluate the proposed heuristics through a set of simulations.

The remainder of this chapter is organized as follows. We first give an overview of related work in Section 2.2. Then we formalize the model in Section 2.3. In Section 2.4, we show that MinMakespan is NP-complete. All the heuristics are presented in Section 2.5, and the experimental evaluation is conducted in Section 2.6. Finally, we summarize our work in Section 2.7.

2.2 Related work

As stated above, rooted trees are commonly used to represent task dependencies for scientific applications. For instance, in dense linear algebra libraries such as SuperMatrix [23] and Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) [20], the dependencies between tasks are well identified, leading to an efficient asynchronous parallel execution of tasks. However, in sparse linear algebra, scheduling trees is more difficult because of enormous tasks' amount and their irregular weights [56]. Liu [63] gives a detailed description of the construction of the elimination tree, its use for Cholesky and LU factorizations and its role in multifrontal methods. In [61], Liu introduces two techniques for reducing the memory requirement in post-order tree traversals. In the subsequent work [62], the post-order constraint is dropped and an efficient algorithm to find a possible ordering for the multifrontal method is given. Building upon Liu's work, [50] proposed a new exact algorithm for exploring a tree with the minimum memory requirement, and studied how to minimize the I/O volume when out-of-core execution is required. The problem of general task graphs handling large data has also been identified by Ramakrishnan et al. [72], who propose some simple heuristics. Their work was continued by Bharathi et al. [14], who

develop genetic algorithms to schedule such workflows.

Several recent studies have considered parallel sparse matrix solvers, and they have investigated techniques and algorithms to reduce communication and execution times on different systems (shared memory, distributed). Kim et al. [56] propose a two-level task parallelism algorithm, which first partitions the tree into many subtrees, and then further decomposes subtrees into regular fine-grained tasks. In this work, the scheduling of executing tasks of the first level is handled by OpenMP dynamically, which however may cause an arbitrarily bad memory consumption. In a later work, Kim et al. [55] take memory bound into consideration through Kokkos's [37] dynamic task scheduling and memory management. Agullo et al. [2] also take advantage of two-level parallelism and discussed the ease of programming and the performance of the program. Targeting at distributed memory systems, Sao et al. [76] partition the tree into two levels, a common ancestor with its children, and then replicate the ancestor to processors that are in charge of children; both communication time and makespan are reduced by this method, at the expense of a larger memory consumption.

Partitioning a tree, or more generally a graph into separate subsets to optimize some metric has been thoroughly studied. Graph partitioning has various applications in parallel processing, complex networks, image processing, etc. Generally, these problems are NP-hard. Exact algorithms have been proposed, which mainly rely on branch-and-bound framework [58], and are appropriate only for very small graphs and small number of resulting subgraphs. A large variety of heuristics and approximation algorithms for this problem have been presented. Some of them directly partition the entire graph, such as spectral partitioning that uses eigenvector from Laplacian matrix to infer the global structure of a graph [34, 33], geometric partitioning that considers coordinates of graph nodes and projection to find an optimal bisecting plane [79, 68], streaming graph partitioning that uses much less memory and time, applied mainly in big data processing [81]. Their results can be iteratively improved by different types of strategies: node-swapping between adjacent subgraphs [54, 42, 75], graphing growing from some carefully selected nodes [31, 77], randomly choosing nodes to visit according to transition probabilities [65]. A multi-level scheme that consists of contraction, partitioning on the smaller graphs and mapping back to the original graph and improvement, can give a high quality results in a short execution time [19]. For a concise review of graph partitioning, see [19].

When focusing on trees rather than general graphs, the balanced partitioning problem is still difficult [39]. It is APX-hard to approximate the cut size within any finite factor if subtrees are strictly balanced, some studies hence approximate the cut size as well as the balance, known as bicriteria-approximation [6]. When near-balance is allowed, tree partitioning is promising. Feldmann and Foschini [40] give a polynomial-time algorithm that cuts no more edges than an optimal perfectly balanced solution.

Compared to the classical graph partitioning studies, which tend towards balanced partitions (subgraphs with approximately the same weight), our problem considers a more complex memory constraint on each component, which makes the previous work on graph partitioning unsuitable to find a good partitioning strategy.

2.3 Model

We consider a tree-shaped task graph τ, where the vertices (or nodes) of the tree, numbered from 1 to n, correspond to tasks, and the edges correspond to precedence constraints among the tasks. The tree is rooted (node r is the root, where $1 \leq r \leq n$), and all precedence constraints are oriented towards the leaves of the tree. Note that we may similarly consider precedence constraints oriented towards the root by reversing all schedules, as outlined in [50]. A precedence constraint $i \rightarrow j$ means that task j needs to receive a file (or data) from its parent i before it can start its execution. Each task i in the rooted tree is characterized

by the size f_i of its input file, and by the size m_i of its temporary execution data (and for the root r, we assume that $f_r = 0$). A task can be processed by a given processor only if all the task's data (input file, output files, and execution data) fit in the processor's currently available memory. More formally, let M be the size of the main memory of the processor, and let S be the set of files stored in this memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. The processing of task i is possible if we have:

$$MemReq(i) = f_i + m_i + \sum_{j \in children(i)} f_j \leq M - \sum_{j \in S, j \neq i} f_j,$$

where $MemReq(i)$ denotes the memory requirement of task i, and $children(i)$ are its children nodes in the tree. Intuitively, M should exceed the largest memory requirement over all tasks (denoted as $MaxOutDeg$ in the following), so as to be able to process each task:

$$MaxOutDeg = \max_{1 \leq i \leq n} (MemReq(i)) \leq M.$$

However, this amount of memory is in general not sufficient to process the whole tree, as input files of unprocessed tasks must be kept in memory until they are processed.

Task i can be executed once its parent, denoted $parent(i)$, has completed its execution, and the execution time for task i is w_i. Of course, it must fit in memory to be executed. If the whole tree fits in memory and is executed sequentially on a single processor, the execution time, or *makespan*, is $\sum_{i=1}^{n} w_i$. In this case, the task schedule, i.e., the order in which tasks of τ are processed, plays a key role in determining how much memory is needed to execute the whole tree in main memory. When tasks are scheduled sequentially, such a schedule is a topological order of the tree, also called a traversal. One can figure out the minimum memory requirement of a task tree τ and the corresponding traversal using the work of Liu [62] or the work of Mathias [50]. We denote by $MinMemory(\tau)$ the minimum amount of memory necessary to complete task tree τ.

The target platform consists of p identical processors, each equipped with a memory of size M. The aim is to benefit from this parallel platform both for memory, by allowing the execution of a tree that does not fit within the memory of a single processor, and also for makespan, since several parts of the tree could then be executed in parallel. The goal is therefore to partition the tree workflow τ into $k \leq p$ parts $\tau_1, \ldots, \tau_k$, which are connected components of the original tree. Hence, each part τ_i is itself a tree. We refer to these connected components as *subtrees* of τ. Note that τ can also be viewed as a tree made of these subtrees. Such a partition is illustrated on Figure 2.1, where the tree is decomposed into five subtrees: τ_1 with nodes 1, 2, and 3; τ_2 with nodes 4, 6, and 7; τ_3 with node 5; τ_4 with node 8; and τ_5 with node 9. We require that each subtree τ_i can be each executed within the memory of a single processor, i.e., $MinMemory(\tau_\ell) \leq M$, for $1 \leq \ell \leq k$. We are to execute such k subtrees on k processors. Let $root(\tau_\ell)$ be the task at the root of subtree τ_ℓ. If $root(\tau_\ell) \neq r$, the processor in charge of tree τ_ℓ needs to receive some data from the processor in charge of the tree containing $parent(root(\tau_\ell))$, and this data is a file of size $f_{root(\tau_\ell)}$. This can be done within a time $\frac{f_{root(\tau_\ell)}}{\beta}$, where β is the available bandwidth between each couple of processors.

We denote by $alloc(i)$ the set of tasks included in subtree τ_ℓ rooted in $root(\tau_\ell) = i$, and by $desc(i)$ the set of tasks, not in $alloc(i)$, that have a parent in $alloc(i)$:

$$desc(i) = \{j \notin alloc(i) \mid parent(j) \in alloc(i)\}.$$

The makespan can then be expressed with a recursive formula. Let $MS(i)$ denote the time (or makespan) required to execute the whole subtree rooted in i, given a partition into subtrees. Note that the whole subtree rooted in i may contain several subtrees of the partition (it is τ for $i = r$). The goal is hence to express $MS(r)$, which is the makespan

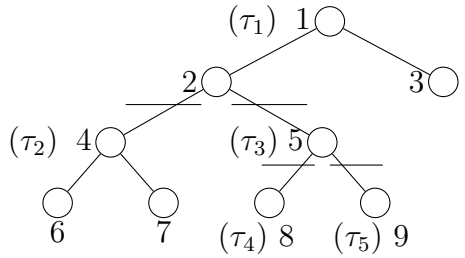


Figure 2.1 – Tree partition and recursive computation of makespan.

of τ. We have (recall that $f_r = 0$ by convention):

$$MS(i) = \frac{f_i}{\beta} + \sum_{j \in alloc(i)} w_j + \max_{k \in desc(i)} MS(k).$$

We assume that the whole subtree τ_ℓ is computed before initiating communication with its children.

The goal is to find a decomposition of the tree into $k \leq p$ subtrees that all fit in the available memory of a processor, so as to minimize the makespan $MS(r)$. Figure 2.1 exhibits an example of such a tree decomposition, where the horizontal lines represent the edges cut to disconnect the tree τ into five subtrees. Subtree τ_1 is executed first, after receiving its input file of size $f_1 = 0$, and it includes tasks 1, 2 and 3. Then, subtrees τ_2 and τ_3 are processed in parallel. The final makespan for τ_1 is thus:

$$MS(1) = \frac{f_1}{\beta} + w_1 + w_2 + w_3 + \max(MS(4), MS(5)),$$

where $MS(5)$ recursively calls $\max(MS(8), MS(9))$, since τ_4 and τ_5 can also be processed in parallel.

For convenience, we also denote by W_i the sum of the weights of all nodes in the subtree rooted in i (hence, for a leaf node, $W_i = w_i$):

$$W_i = w_i + \sum_{j \in children(i)} W_j.$$

We are now ready to formalize the optimization problem that we consider:

Definition 1 (MinMakespan)**.** *Given a task tree τ with n nodes, a set of p processors each with a fixed amount of memory M, partition the tree into $k \leq p$ subtrees $\tau_1, \ldots, \tau_k$ such that $MinMemory(\tau_i) \leq M$ for $1 \leq i \leq k$, and the makespan is minimized.*

Given a tree τ and its partition into subtrees τ_i , we consider its quotient graph Q given by the partition: vertices from a same subtree are represented by a single vertex in the quotient tree, and there is an edge between two vertices $u \to v$ of the quotient graph if and only if there is an edge in the tree between two vertices $i \to j$ such that $i \in \tau_u$ and $j \in \tau_v$. Note that since we impose a partition into subtrees, the quotient graph is indeed a tree. This quotient tree will be helpful to compute the makespan and to exhibit the dependencies between the subtrees.

2.4 Problem complexity

Theorem 1. *The (decision version of)* MinMakespan *problem is NP-complete.*

Proof. First, it is easy to check that the problem belongs to NP: given a partition of the tree into $k \leq p$ subtrees, we can check in polynomial time that (i) the memory needed for

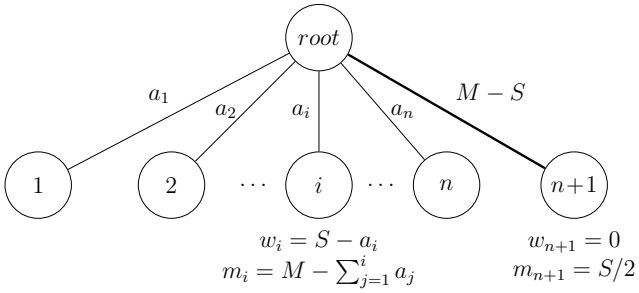


Figure 2.2 – Tree of instance $\mathcal{I}_2$ used in the NPC proof.

each subtree does not exceed M, and that (ii) the obtained makespan is not larger than a given bound.

To prove the completeness, we use a reduction from 2-partition [43]. We consider an instance $\mathcal{I}_1$ of 2-partition: given n positive integers $a_1, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S/2$, where $S = \sum_{i=1}^{n} a_i$. We consider the 2-partition-equal variant of the problem, also NP-complete, where both partitions have the same number of elements ($|I| = n/2$, and thus, n is even). Furthermore, we assume that $n \geq 4$, since the problem is trivial for $n = 2$. From $\mathcal{I}_1$, we build an instance $\mathcal{I}_2$ of MinMakespan as follows:

- The tree τ consists of $n+2$ nodes, and it is described on Figure 2.2: it is a fork graph (a root with $n+1$ children). The weights on edges represent the size of input files f_i, and the computation time and memory requirements are indicated respectively by w_i and m_i ($0 \leq i \leq n+1$, where $root = 0$).
- For $1 \leq i \leq n$, $w_i = S - a_i$, $m_i = M - \sum_{j=1}^{i} a_j$, and $f_i = a_i$.
- For the last child, $w_{n+1} = 0$, $m_{n+1} = S/2$, and $f_{n+1} = M - S$.
- For the root, $w_{root} = 0$, $m_{root} = 0$, and $f_{root} = 0$.
- The makespan bound is $C_{\max} = (n+1)\frac{S}{2}$.
- The memory bound is $M = C_{\max} + S + 1$.
- The bandwidth is $\beta = 1$.
- The number of processors is $p = \frac{n}{2} + 1$.

Consider first that $\mathcal{I}_1$ has a solution, I, such that $I \subseteq \{1, \ldots, n\}$ and $|I| = n/2$ (i.e., I contains exactly $n/2$ elements). We execute sequentially root and task $n+1$, plus the tasks in I, and we pay communications and execute in parallel tasks not in I. We can execute each of these tasks in parallel since there are $n/2+1$ processors and exactly $n/2$ tasks not in I. Since we have cut nodes not in I, there remains exactly files of size $S/2$ in memory, plus $f_{2n+1} = M - S$, and to execute task $n+1$, we also need to accommodate $m_{2n+1} = S/2$, hence we use exactly a memory of size M. We can then execute nodes in I starting from the right of the tree, without exceeding the memory bound. Indeed, once task $n+1$ has been executed, there remains only some of the $f_i = a_i$'s in memory, and they fit together with m_i in memory. The makespan is therefore $\frac{n}{2}S - \frac{S}{2}$ for the sequential part (executing all tasks in I), and each of the tasks not in I can be executed within a time S (since $\beta = 1$), all of them in parallel, hence a total makespan of $(n-1)\frac{S}{2} + S = C_{\max}$. Hence, $\mathcal{I}_2$ has a solution.

Consider now that $\mathcal{I}_2$ has a solution. First, because of the constraint on the makespan, $root$ and task $n+1$ must be in the same subtree, otherwise we would pay a communication of $M - S = C_{\max} + 1$, which is not acceptable. Let I be the set of tasks that are executed on the same subtree as root and task $n+1$. I contains at least $\frac{n}{2}$ tasks, since the number of

processors is $\frac{n}{2}+1$. If I contains more than $\frac{n}{2}$ tasks, then the makespan is strictly greater than $(\frac{n}{2}+1)S - S$ for the sequential part, plus S for all other tasks done in parallel, that is $(\frac{n}{2}+1)S > C_{\max}$. Therefore, I contains exactly $\frac{n}{2}$ tasks.

The constraint on makespan requires that $\frac{n}{2}S - \sum_{i\in I} a_i + S \leq C_{\max}$, and hence $\sum_{i\in I} a_i \geq \frac{S}{2}$. After executing *root*, the files remaining in memory are the files from tasks in I and f_{n+1}, since other files are communicated to other processors. As long as f_{n+1} is in memory, no task of I can be executed due to the memory constraint, hence to execute task $n+1$, the memory constraint writes $\sum_{i\in I} a_i + M - S + \frac{S}{2} \leq M$, hence $\sum_{i\in I} a_i \leq \frac{S}{2}$. Therefore, we must have $\sum_{i\in I} a_i = \frac{S}{2}$, and we have found a solution to $\mathcal{I}_1$. □

2.5 Heuristic strategies

In this section, we design polynomial-time heuristics to solve the MinMakespan problem. The heuristics work in three steps: (1) partition the tree into subtrees in order to minimize the makespan, without accounting for the memory constraint; (2) partition subtrees that do not fit in memory, i.e., such that $\mathit{MinMemory}(\tau_i) > M$; (3) ensure that we do have the correct number of subtrees, i.e., merge some subtrees if there are more subtrees than processors, or further split subtrees if there are extra processors and the makespan can be reduced. We now detail the three steps, focusing on makespan, then memory, then number of processors.

2.5.1 Step 1: Minimizing the makespan

In the first step, the objective is to split the tree into a number of subtrees, each processed by a single processor, in order to minimize the makespan. We will consider the memory constraint on each subtree at the next step (Section 2.5.2).

We first consider the case where the tree is a linear chain, and prove that its optimal solution uses a single processor.

Lemma 1. *Given a tree τ such that all nodes have at most one child (i.e., it is a linear chain), the optimal makespan is obtained by executing τ on a single processor, and the optimal makespan is $\sum_{i=1}^{n} w_i$.*

Proof. If more than one processor is used, all tasks are still executed sequentially because of dependencies, but we further need to account for communicating the f_i's between processors. Therefore, the makespan can only be increased. □

More generally, if the decomposition into subtrees form a linear chain, as defined below, then the subtrees must be executed one after the other, no parallelism is exploited and unnecessary communication may occur.

Definition 2 (Chain)**.** *Given a tree τ, its partition into subtrees τ_i and the resulting quotient tree Q, a chain is a set of nodes $u_1, \dots u_k$ of Q such that u_i is the only child of u_{i-1} ($i > 1$).*

Therefore, having several subtrees as a linear chain can only increase the makespan, compared to an execution of the whole tree on a single processor.

We now propose four heuristics that aim at minimizing the makespan, and hence avoid having linear chains of subtrees.

Two-level heuristic

The first heuristic, SplitSubtrees is adapted from [38], where the goal was to reduce the makespan while limiting the memory in a shared-memory environment. It creates a two-level partition with one connected component containing the root, executed first on a

Algorithm 1 SplitSubtrees (τ, p)

for all nodes $i \in \tau$ **do**
  $MS(i) = W_i + \frac{f_i}{\beta}$;
end for
$PQ_0 \leftarrow \{r\}$; (the priority queue consists of the tree root)
$seqSet \leftarrow \emptyset$;
$MS_0 = MS(r)$;
$s \leftarrow 1$; (splitting rank)
while $head(PQ_{s-1})$ is not a leaf **do**
  $i \leftarrow popHead(PQ_{s-1})$;
  $seqSet \leftarrow seqSet \cup \{i\}$;
  $PQ_s \leftarrow PQ_{s-1} \setminus \{i\} \cup children(i)$;
  if $|PQ_s| > p - 1$ **then**
    Let $\mathcal{S}$ denote the $|PQ_s| - (p-1)$ smallest nodes, in terms of W_i, in PQ_s;
  else
    $\mathcal{S} = \emptyset$;
  end if
  $MS_s = \sum_{i \in seqSet} w_i + \sum_{i \in \mathcal{S}} W_i + \max_{k \in PQ_s \setminus \mathcal{S}}(MS(k))$;
  $s \leftarrow s + 1$;
end while
select splitting s^* that leads to the smallest MS_{s^*};
return PQ_{s^*};

single processor (and called the *sequential set*), followed by the parallel processing of $p-1$ independent subtrees. In the context of shared memory, this heuristic has been proven the two-level partition with best makespan [38, Lemma 5.1]. We adapt it to our context, in order to take communications into account.

The SplitSubtrees heuristic relies on a splitting algorithm, which maintains a set of subtrees and iteratively *splits* the subtree with the largest makespan. Initially, the only subtree is the whole tree. When a subtree is split, its root is moved to the sequential set (denoted *seqSet*) and all its children subtrees are added to the current set of subtrees. Algorithm 1 formalizes the heuristic, in which the current set of subtrees is stored in a priority queue PQ sorted by non-increasing makespan: $MS(i) = W_i + \frac{f_i}{\beta}$ (which accounts for communications). We assume that the first element of PQ is always the element that has the greatest makespan.

For a given state of the algorithm (i.e., a partition of the tree between *seqSet* and subtrees in PQ), we consider the following mapping: the $p-1$ largest subtrees in PQ (in terms of total computation weight W) are allocated to distinct processors, while the remaining subtrees are processed by the same processor in charge of the sequential set. Note that all these nodes (*seqSet* plus the smallest subtrees of PQ) form a connected component of the original tree: *seqSet* is a connected component containing the root, and each root of a subtree in PQ has its parent in *seqSet*.

We iteratively consider the solutions obtained by the successive splitting operations and finally select the one with the best makespan. We stop splitting subtrees when the largest subtree in PQ is indeed a leaf. Thus, there are at most n iterations, hence the algorithm is polynomial. The algorithm returns the set of nodes that are the root of a subtree, which corresponds to a *cut* of the tree, i.e., the set of edges that are cut to partition the tree into subtrees.

Figure 2.3 – Two cases where SplitSubtrees is suboptimal. Dashed edges represent the solution of SplitSubtrees, plain edges give the optimal partition.

Improving the SplitSubtrees heuristic

There are two main limitations of SplitSubtrees. First, it produces only a two-level solution: in the provided decomposition, all subtrees except one are the children of the subtree containing the root. In some cases, as illustrated on Figure 2.3, it is beneficial to split the tree into more levels. In these examples, we have $p = 7$ processors. Node labels denote their computational weights (10 for all nodes, except three of them per tree), and there are no communication costs. The horizontal dashed lines represent the edges cut in the solution of SplitSubtrees, while solid lines represent the optimal partition. On the example of Figure 2.3(a), a two-level solution cannot achieve a makespan better than 40. If the cut was made at a lower level, the makespan would be even greater. It is however possible to achieve a makespan of 33 by cutting at two levels.

The second limitation is the possibly too large size of the first subtree, containing the sequential set *seqSet*. Since its execution is sequential, it may lead to a large resource waste. This is for instance the case in the example of Figure 2.3(b), where the optimal two-level solution has a sequential set whose execution time is 31, while further parallelism could have been used: the optimal solution cuts this sequential set in order to minimize the makespan.

To address these limitations, we design a new heuristic, ImprovedSplit (see Algorithm 2), which improves upon SplitSubtrees by building a multi-level solution. Since we aim at further cutting the tree to obtain a multi-level solution, ImprovedSplit does not set a limit on the number of subtrees in a first step, but rather tries to create as many subtrees as possible, while the makespan can be improved. It is initially called with $T = \tau$, and first calls SplitSubtrees with no restriction on the number of subtrees: p is set to $+\infty$. Then, ImprovedSplit recursively tries to split the sequential set and the largest children subtrees (subtrees whose roots are in PQ), until the makespan cannot be further reduced (again, with no restriction on the number of subtrees).

Finally, once all splits have been done, if there are more subtrees than processors, some of them are merged with a call to Merge (which will be explained in Section 2.5.3), without accounting for the memory constraint (call with infinite memory). The use of *AlreadyOptSet* ensures that ImprovedSplit is called at most once on each node. The makespan computation in the repeat loop has a complexity in $O(n)$, and the loop has at most n iterations. Therefore, we get a complexity in $O(n^2)$ for a call to ImprovedSplit, without the final call to Merge, hence a complexity in $O(n^3)$ for the n calls. Note that Merge does not do anything when $p = +\infty$, since there are enough processors, and during each recursive call to ImprovedSplit, Merge is called with $p = +\infty$, hence has no effect. The complexity of the final Merge is in $O(n^3)$, as we do not consider the memory constraint (see Section 2.5.3). The final complexity of ImprovedSplit is thus $O(n^3)$.

ASAP heuristic

The main idea of this heuristic is to parallelize the processing of tree τ as soon as possible, by cutting edges that are close to the root of the tree. ASAP uses a node priority queue PQ to store all the roots of subtrees produced. Nodes in PQ are sorted by non-increasing W_i's

Algorithm 2 ImprovedSplit (T, p)

```
1: PQ ← SplitSubtrees(T, +∞);
2: AlreadyOptSet ← ∅;
3: τ_i is the subtree of T rooted in i;
4: τ_seq = T \ ∪_{i∈PQ} {τ_i};
5: C_p ← ∅; C_temp ← ∅;
6: repeat
7:    i ← popHead(PQ); W ← MS(i);
8:    if i ∈ AlreadyOptSet then break;
9:    C_temp ← ImprovedSplit(τ_i, +∞); (partition subtrees of parallel parts)
10:   Add i to AlreadyOptSet;
11:   Recompute MS(i) with the new cut C_temp;
12:   if MS(i) < W then C_p ← C_p ∪ C_temp;
13:   Insert i into PQ (sorted by non-increasing makespan);
14: until MS(i) ≥ W or head(PQ) = i
15: C_s ← ImprovedSplit(τ_seq, +∞); (partition seq. set)
16: C ← PQ ∪ C_p ∪ C_s ;
17: if p < |C| + 1 then Merge(T, C, p, +∞);
18: return C
```

(recall that W_i is the total computation weight of the subtree rooted at node i). Iteratively, the heuristic cuts the largest subtree, if it has siblings, until there are as many subtrees as processors (see Algorithm 3 for details). Therefore, it creates a multi-level partition of the tree. It selects the partition that has the minimum makespan.

At this point, we might have chains of subtrees (as defined above), which increases the makespan compared to a sequential execution of these subtrees. Figure 2.4 provides an example where this happens: the makespan is 11+2+12+2+10+(2+10) = 49, since the three leaf tasks of weight 10 are executed in parallel. Four units of communication time could however be saved by executing all other nodes on the same processor, reaching a makespan of 45 and using only four processors.

To avoid this shortcoming, ASAP then builds the quotient tree in which, except the root, other nodes that have no siblings are elements of chains. Their input edges are therefore restored, i.e., subtrees are merged into a single subtree so that there are no more chains, and therefore, this leaves some processors idle. These idle processors will be used, if possible, to improve the makespan, during the last step of the heuristics, see Section 2.5.3.

2.5.2 Step 2: Fitting into memory

After partitioning a tree into many subtrees by SplitSubtrees, ImprovedSplit, or ASAP, we propose three heuristics in this section to check each subtree's minimum memory requirement and further partition those such that $MinMemory(\tau_i) > M$.

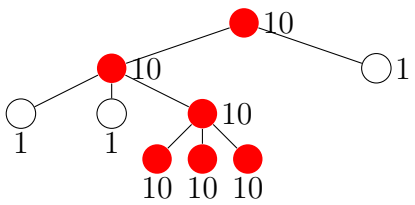


Figure 2.4 – Example showing that a chain always wastes processors. Node labels represent their weight. All edges have weight 2, and $p = 6$. Red nodes denote subtrees' roots as determined by ASAP.

Algorithm 3 ASAP (τ, p)

$PQ \leftarrow$ children of the root of τ, sorted by non-increasing W_i's;
$s = 0$; $C_s \leftarrow \{\text{root of } \tau\}$; ($s$ is the step)
Let MS_s be the makespan of τ with partition C_s;
repeat
  if PQ is empty **then** break;
  $i \leftarrow popHead(PQ)$;
  insert $Children(i)$ into PQ;
  if i is not the only child of its parent **then**
    $s \leftarrow s + 1$;
    $C_s \leftarrow C_{s-1} \cup \{i\}$; (the edge we just cut)
    Let MS_s be the makespan of τ with partition C_s;
  end if
until $|C_s| = p$;
select step s^* that minimizes MS_{s^*};
construct the quotient tree Q from τ and C_{s^*};
for all nodes i of Q **do**
  if node i has only one child **then**
    remove input edge of i's child from C_{s^*};
  end if
end for
return C_{s^*};

FirstFit heuristic

We first note the proximity of this problem with the MinIO problem [50]. In this problem, a similar tree has to be executed on a single processor with limited memory. When the memory shortage happens, some data have to be evicted from the main memory and written to disk. The goal is to minimize the total volume of the evicted data while processing the whole tree. In [50], six heuristics are designed to decide which files should be evicted. In the corresponding simulations, the FirstFit heuristic demonstrated better results. It first computes the traversal (permutation σ of the nodes that specifies their execution sequence) that minimizes the peak memory, using the provided MinMemory algorithm [50]. Given this traversal, if the next node to be processed, denoted as j, is not executable due to memory shortage, we have to evict some data from the memory to the disk. The amount of freed memory should be at least $Need(j) = (MemReq(j) - f_j) - M^{avail}$, where M^{avail} is the currently available memory when we try to execute node j. In that case, FirstFit orders the set $S = \{f_{i_1}, f_{i_2}, \ldots, f_{i_j}\}$ of the data already produced and still residing in main memory, so that $\sigma(i_1) > \sigma(i_2) > \cdots > \sigma(i_j)$, where $\sigma(i)$ is the step of processing node i in the traversal (f_{i_1} is the data that will be used for processing the latest) and selects the first data from S until their total size exceeds or equals $Need(j)$.

We consider the simple adaptation of FirstFit to our problem: the final set of data F that are evicted from the memory defines the edges that are cut in the partition of the tree, thus resulting in $|F| + 1$ subtrees. This guarantees that each subtree can be processed without exceeding the available memory, but may lead to numerous subtrees.

LargestFirst heuristic

For our problem, we want to end up with a total of not more than p subtrees from the original tree (one subtree per processor), and since we may have already created p subtrees in Step 1 (Section 2.5.1), we do not want to create too many additional subtrees. Otherwise, subtrees will have to be merged in Step 3 (Section 2.5.3), possibly resulting in an increase of makespan. Therefore, we propose a variant of the FirstFit strategy, which orders the

set S of candidate data to be evicted by non-increasing sizes f_i, and selects the largest data until their total size exceeds the required amount. This may result into edges with larger weights being cut, and thus an increased communication time, but it is likely to reduce the number of subtrees. This heuristic is called LargestFirst.

Immediately heuristic

Finally, we propose a third and last heuristic to partition a tree into subtrees that fit into memory. As for the previous heuristic, we start from a minimum memory sequential traversal σ. We simulate the execution of σ, and each time we encounter a node that is not executable because of memory shortage, we cut the corresponding edge and this node becomes the root of a new subtree. We continue the process for the remaining nodes, and then recursively apply the same procedure on all created subtrees, until each of them fit in memory. This heuristic is called Immediately.

2.5.3 Step 3: Reaching an acceptable number of subtrees

Now that we have first minimized the makespan, and then made sure that each subtree fits in local memory, we need to check how many subtrees have been generated. During this step, we either decrease the number of subtrees if it is greater than the number of processors p, or we increase it by further splitting subtrees if we have idle processors and the makespan may be improved.

Decreasing the number of subtrees

If there are more subtrees than processors, some of them have to be merged, and the resulted subtrees should also fit in local memory.

For subtrees that are leaves and have only one sibling, merging only theirselves to their parents will lead to a chain, which wastes processors. Thus, they are also merged with their siblings. In all combinations that fit in memory, we greedily merge subtrees that lead to the minimum increase in makespan. We compute the increase in makespan as follows. We denote the subtree to be merged as node i of the quotient tree. Sometimes (when i is not on the critical path), $MS(i)$ can be increased without changing the final makespan $MS(r)$. We define d_i as the *slack* in $MS(i)$, that is, the threshold such that $MS(r)$ is not impacted by the increase of $MS(i)$ up to $MS(i) + d_i$. It can be recursively computed from the root: $d_i = d_t + MS(k) - MS(i)$, in which t is i's parent in the quotient tree and k is the sibling of i that has the maximum makespan. For the root, d_r is set to 0.

We then compute the increase of makespan of merging i to its parent t in the quotient tree as follows. We first estimate the increase Δ_t of $MS(t)$. If i is a leaf and has only one sibling, denoted j, the increase in makespan of their parent t is $\Delta_t = W_i + W_j - \max(\frac{f_i}{\beta} + w_i, \frac{f_j}{\beta} + w_j)$. For other subtrees, the makespan of t before the merge is $MS(t) = \frac{f_t}{\beta} + W_t + \max(MS(k), MS(i))$, and after merging i to t, $MS(t) = \frac{f_t}{\beta} + W_t + W_i + \max(MS(k), MS(j))$, where j is the child of i that has the maximum makespan. Therefore, the increase of $MS(t)$ is $\Delta_t = W_i + \max(MS(k), MS(j)) - \max(MS(k), MS(i))$. Finally, taking the slack into consideration, the increase of $MS(r)$ is $\Delta_t - d_i$.

This algorithm is formalized as Algorithm 4. There are initially at most n subtrees, and we decrease this number by one or two at each step, hence the algorithm runs in polynomial time.

Increasing the number of subtrees

If there are more processors than subtrees, we may be able to further reduce the makespan by splitting some of the subtrees. Given a tree τ and a partition C, SplitAgain first builds

Algorithm 4 MERGE (τ, C, p, M)

```
1: Construct the quotient tree Q according to τ and C;
2: shortage ← p − |C| − 1; (amount of processors' shortage)
3: r ← root of τ;
4: while shortage > 0 do
5:    for all nodes i of Q except the root do
6:       if subtree i is a leaf and has only one sibling then
7:          Δ_i ← estimation of increase in MS(r) if subtree i and its sibling are merged with
            their parent;
8:          m_i ← subtree made of i, its sibling and their parent fits in memory size M;
9:       else
10:         Δ_i ← estimation of increase in MS(r) if merge subtree i with its parent;
11:         m_i ← subtree made of i and its parent fits in memory size M;
12:      end if
13:   end for
14:   set S ← {i s.t. m_i = true};
15:   j ← combination in S that has the minimum Δ_i;
16:   if subtree j is a leaf and has only one sibling then
17:      merge subtree j and its sibling with their parent; shortage = shortage − 2;
18:   else
19:      merge subtree j with its parent;
         shortage = shortage − 1;
20:   end if
21: end while
```

the quotient tree Q to model dependencies among subtrees, and finds its critical path. A critical path is a set of nodes of Q that defines the makespan of τ. In the example of Figure 2.5, the critical path consists of three nodes of the quotient tree. Each subtree on the critical path is a candidate to be cut into two (or three) parts by cutting some edges. The set L (black nodes in Figure 2.5) contains the nodes whose input edge could be cut. If the subtree is a leaf in the quotient tree, we always split into three parts, otherwise we would create a chain and only increase the makespan. At each step, we greedily select the option (within nodes of L) that has the maximum potential decrease in makespan of τ. We compute the potential makespan decrease as follows: let i be the node whose input edge is considered to be cut. It currently lies in the subtree τ_t rooted at node t. After cutting the input edge of node i, it produces a new subtree τ_i of weight W_i.

The makespan of τ_t after cutting is given by:

$$\max(MS(t) - W_i, W_t + \frac{f_i}{\beta} + \max_{\tau_j \in Children(\tau_i)} MS(j)),$$

where $MS(j)$ is the makespan of a subtree rooted at j before cutting any new edge. Indeed,

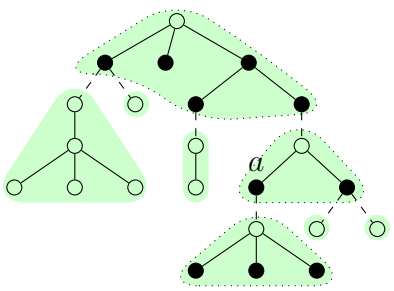


Figure 2.5 – Example to illustrate SPLITAGAIN: green areas surrounded with dotted line belong to the critical path; black nodes are candidates to be cut after line 5 (set L).

either the critical path does not include τ_i, or it now includes the communication to τ_i and the makespan of the largest children of τ_i in the new quotient tree. Note that if the child of τ_t that is in the critical path is also a child of τ_i (for instance, in Figure 2.5, when we try to cut the input edge of node a), the makespan will only be increased, and hence we will never cut edge i.

The decrease of the makespan of τ_t when cutting the input edge of node i is thus given by:

$$\Delta_i = \min(W_i, MS(t) - W_t - \frac{f_i}{\beta} - \max_{\tau_j \in Children(\tau_i)} MS(j)).$$

If we cut two edges in the last subtree on the critical path, say i and j, the makespan after cutting is

$$MS(t) - W_i - W_j + \max(\frac{f_i}{\beta} + W_i, \frac{f_j}{\beta} + W_j),$$

and the decrease of $MS(t)$ is:

$$\Delta_i = \min(W_j - \frac{f_i}{\beta}, W_i - \frac{f_j}{\beta}).$$

This process is repeated until there are no more idle processors or no further decrease in makespan. It is formalized in Algorithm 5.

Algorithm 5 SplitAgain(τ, C, p)

```
1: Compute the quotient tree Q and its critical path CriPat;
2: n ← p − 1 − |C|; (number of idle processors)
3: while n ≥ 1 do
4:    L ← nodes of subtrees on CriPat;
5:    Remove from L the roots of subtrees;
6:    for all nodes i in L do
7:       if i is in the last subtree on CriPat and n ≥ 2 then
8:          Let j be the largest sibling of i, in terms of W;
9:          C_i ← input edges of nodes i and j;
10:      else
11:         C_i ← input edge of node i;
12:      end if
13:      Δ_i ← makespan decrease when edges in C_i are cut;
14:   end for
15:   k ← the node in L which leads to the largest Δ_k;
16:   if Δ_k ≥ 0 then
17:      C ← C ∪ C_k; (cut edges in C_k)
18:      n ← n − |C_k|;
19:      Recompute Q and CriPat;
20:   else
21:      break;
22:   end if
23: end while
```

2.6 Experimental validation through simulations

In this section, we compare the performance of the proposed heuristics on a wide range of computing platform settings. We evaluate the results of the three stages: partition for reducing makespan, fitting in the memory constraint and constraint on the number of processors.

2.6.1 Dataset and simulation setup

The dataset contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection. We selected square matrices, whose number of rows is between 2×10^4 and 10^6, and whose number of non-zeros per row is at least 2.5, and the total number of non-zeros is at most 5×10^6. These 76 matrices were first ordered using AMD or MeTiS, then the corresponding elimination trees were built, and relaxed node amalgamation was performed on these trees (see [50] for more details on this construction).

To test the heuristics proposed above, we only kept trees whose MinMemory is larger than its MaxOutDeg. This corresponds to 31 trees in the data set, coming from 22 matrices.

To compare the performance of the proposed heuristics in different environments, and since these trees exhibit very different number of nodes (from thousands to several millions), we have selected three different processor to node ratios (PNR): the number of processors p can be set to $1e-04$, 0.001, or 0.01 times the tree size n (while ensuring $p \geq 3$). We also consider three scenarios for the relative cost of computations vs. communications. Given a tree, we select the communication bandwidth β such that the average communication to computation ratio (CCR), defined as the total time for communicating all data divided by the total computation time, is either 0.1, 1 or 10.

We consider two scenarios for the memory constraint: (i) in the *loose* scenario, the memory bound for each processor is set to *MinMemory*, hence there is no memory constraint; (ii) then, the *strict* scenario sets the memory bound to *MaxOutDeg*, the minimum memory needed to process any single task. The sequential tree traversal used in FirstFit, LargestFirst and Immediately is given by *MinMem* as described in [50], which has a minimum memory cost. All codes and trees can be found on github.com/gouchangjiang.

2.6.2 Step 1: Minimizing the makespan

The results of heuristics for reducing makespan on different computing scenarios are shown on Figure 2.6. We consider all combinations of CCRs and PNRs, and we normalize the makespan of SplitSubtrees, ImprovedSplit, and ASAP to the makespan obtained with a sequential execution of the tree, denoted by Sequence. Hence, a smaller ratio indicates a better relative performance. Note that there is no memory constraint in this step, hence Sequence returns a valid solution, using only one processor.

As expected, all heuristics are better than the reference sequential schedule Sequence: the makespan is reduced by at least 45% on more than 50% of the cases. With more processors, they behave even better than Sequence, four times better on more than 50% of the cases. Increasing the number of processors generally allows to reduce the makespan, except for SplitSubtrees. All heuristics behave better than SplitSubtrees for all CCR values. Also, note that ImprovedSplit always surpasses ASAP with few processors (PNR=$1e-04$ or 0.001).

Figure 2.6 presents the number of subtrees that are generated compared to the number of processors provided. Only ImprovedSplit takes fully advantage of processor resources in all cases. SplitSubtrees uses all processors only with few processors (PNR=$1e-04$) and not with more processors because it only splits in two levels. For instance, for PNR=0.01, SplitSubtrees uses 16% of the processors on more than 50% of the cases. ASAP uses

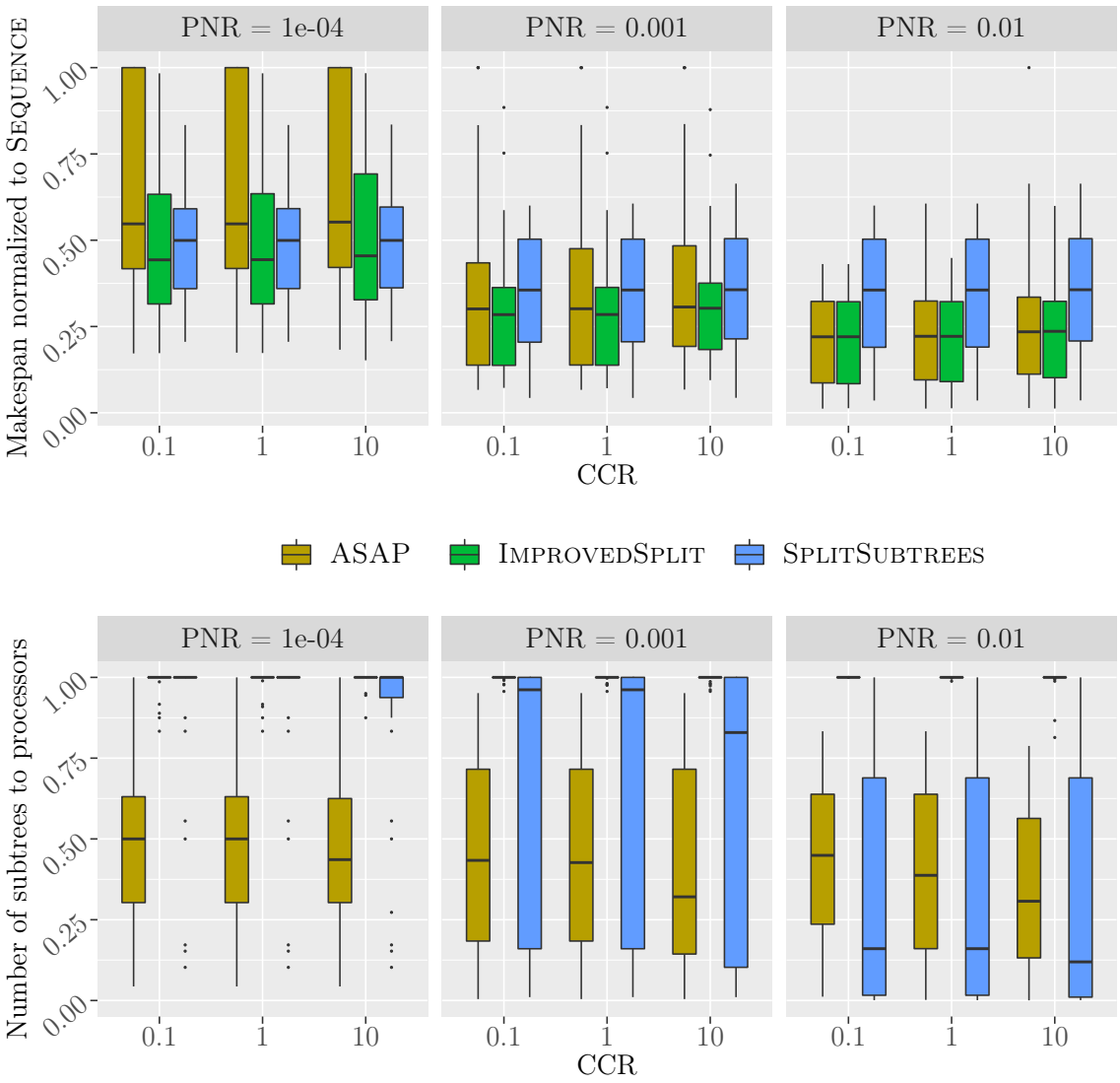


Figure 2.6 – Makespan (*top*, normalized to Sequence) and number of generated subtrees (*bottom*) after Step 1 with different CCRs and PNRs.

much fewer processors than ImprovedSplit, using only half of the processors in around 50% of the cases.

2.6.3 Step 2: Fitting into memory

At the end of Step 1, some subtrees may exceed the maximum available memory M when we consider the *strict* memory scenario. As expected, there are less subtrees not fitting into memory when there are many processors, since subtrees are smaller, and also when using ImprovedSplit, since it generates more subtrees, and hence smaller subtrees. The subtrees that do not fit into memory are further decomposed with either FirstFit, LargestFirst or Immediately, so that all subtrees fit in memory at the end of this step. We may then have more subtrees than processors, and Step 3 will later merge subtrees if needed.

In order to assess the performance of the heuristics from Step 2, we execute them in the *strict* memory scenario both on the original tree (Sequence, i.e., no heuristic from Step 1 is used) and after running the heuristics from Step 1. We report the average ratio of number of subtrees to processors NtoP, and the average percentage of gain on execution time

$$\text{ET} = 100 \times \left(1 - \frac{MS^2_\infty}{MS^1}\right),$$

where MS^2_∞ is the makespan after Step 2 with an infinite number of processors (since

CCR	Heuristic		PNR					
			1e-04		0.001		0.01	
			NtoP	ET	NtoP	ET	NtoP	ET
0.1	FirstFit	ASAP	1.34	7%	0.52	4%	0.42	0%
		ImprovedSplit	1.81	7%	1.10	2%	1.00	0%
		SplitSubtrees	1.66	8%	0.70	0%	0.33	0%
		Sequence	1.49	13%	0.20	13%	0.02	13%
	LargestFirst	ASAP	1.59	8%	0.52	4%	0.42	0%
		ImprovedSplit	2.03	8%	1.10	0%	1.00	0%
		SplitSubtrees	1.86	10%	0.71	0%	0.33	0%
		Sequence	2.17	13%	0.28	13%	0.03	13%
	Immediately	ASAP	5.97	9%	0.91	6%	0.43	2%
		ImprovedSplit	5.55	5%	1.39	0%	1.00	0%
		SplitSubtrees	5.13	9%	0.97	4%	0.33	0%
		Sequence	6.24	18%	0.90	18%	0.09	18%
1	FirstFit	ASAP	1.34	7%	0.51	4%	0.40	0%
		ImprovedSplit	1.81	7%	1.09	2%	1.00	0%
		SplitSubtrees	1.66	8%	0.70	0%	0.33	0%
		Sequence	1.49	13%	0.20	13%	0.02	13%
	LargestFirst	ASAP	1.59	8%	0.51	4%	0.40	0%
		ImprovedSplit	2.07	9%	1.10	4%	1.00	0%
		SplitSubtrees	1.86	10%	0.71	0%	0.33	0%
		Sequence	2.17	13%	0.28	13%	0.03	13%
	Immediately	ASAP	5.97	9%	0.90	6%	0.41	2%
		ImprovedSplit	5.61	5%	1.38	1%	1.00	0%
		SplitSubtrees	5.13	9%	0.97	4%	0.33	0%
		Sequence	6.24	18%	0.90	18%	0.09	18%
10	FirstFit	ASAP	1.34	6%	0.48	3%	0.34	-1%
		ImprovedSplit	1.75	10%	1.09	4%	0.99	0%
		SplitSubtrees	1.64	7%	0.67	-1%	0.33	-1%
		Sequence	1.49	12%	0.20	12%	0.02	12%
	LargestFirst	ASAP	1.97	7%	0.48	3%	0.34	-1%
		ImprovedSplit	2.02	11%	1.10	7%	0.99	0%
		SplitSubtrees	1.84	10%	0.68	-1%	0.33	-1%
		Sequence	2.17	12%	0.28	12%	0.03	12%
	Immediately	ASAP	5.97	8%	0.87	5%	0.35	2%
		ImprovedSplit	6.08	4%	1.38	1%	0.99	0%
		SplitSubtrees	5.11	8%	0.94	3%	0.33	0%
		Sequence	6.24	17%	0.90	17%	0.09	17%

Table 2.1 – After Step 2, NtoP is the ratio of number of subtrees to processors, and ET is the gain on execution time.

splitting subtrees in Step 2 may generate more subtrees than available processors), and MS^1 is the makespan after Step 1. If ET is positive, it means that the new makespan is better, and the partition is feasible only if NtoP is smaller than or equal to 1.

Table 2.1 presents all results, for the three heuristics of Step 2 (FirstFit, LargestFirst and Immediately) and the four possibilities for Step 1. Overall, FirstFit generates the smallest amount of subtrees, hence there is more chance that this heuristic will succeed to map all subtrees to processors while fitting in memory. LargestFirst has close results in terms of subtrees, and it is interesting to see that it can reduce the makespan even more than FirstFit. Immediately generates much more subtrees, and in some cases it is able to further decrease the makespan, but not always. The execution time of FirstFit and LargestFirst could be worse than makespan of Step 1 when communication is expensive (i.e. CCR=10), since creating additional subtrees to fit into memory may generate expensive communications. In conclusion, FirstFit is the best choice when processors are limited (PNR=$1e-04$), LargestFirst and Immediately are also good options when many processors (PNR$\geq$ 0.001) are available.

In the *loose* scenario, all subtrees fit in memory and hence Step 2 does not do anything, and in the following, we apply LargestFirst, FirstFit or Immediately for Step 2 seperately in the *strict* scenario.

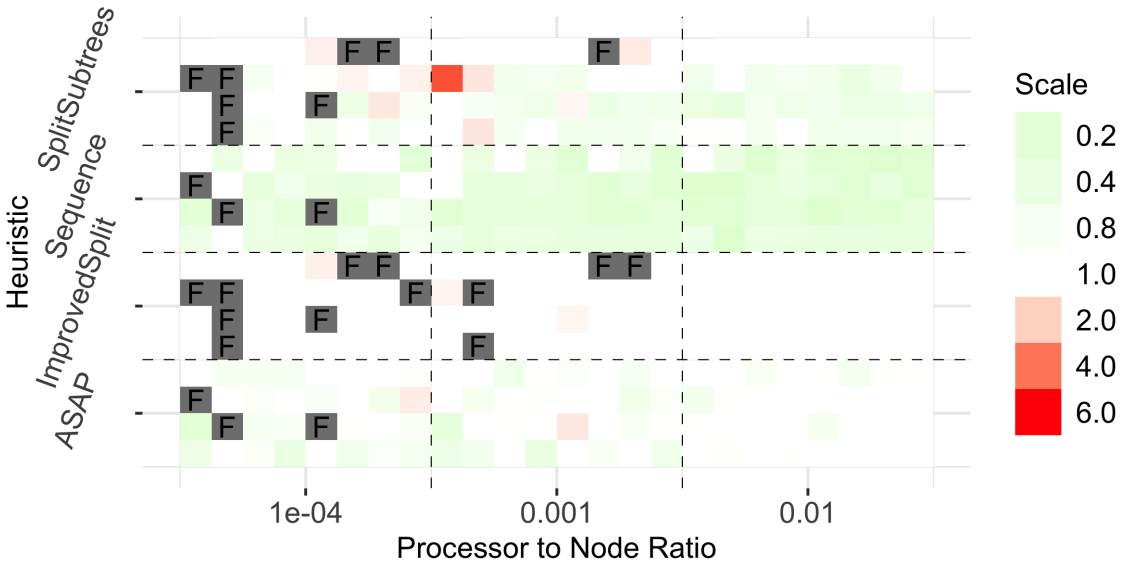
2.6.4 Step 3: Reaching an acceptable number of subtrees

In this section, we examine the performance of Merge and SplitAgain, which are designed for reducing the number of subtrees so that we have enough processors, or for further optimizing the makespan if there are some remaining processors. As seen in Table 2.1, ImprovedSplit is the heuristic generating the most subtrees when combined with LargestFirst, and hence it requires to merge some subtrees to obtain a feasible solution. The other heuristics leave many processors idle when there are many processors (PNR=0.001 or PNR=0.01), and we may be able to further improve the makespan by using SplitAgain in these cases.

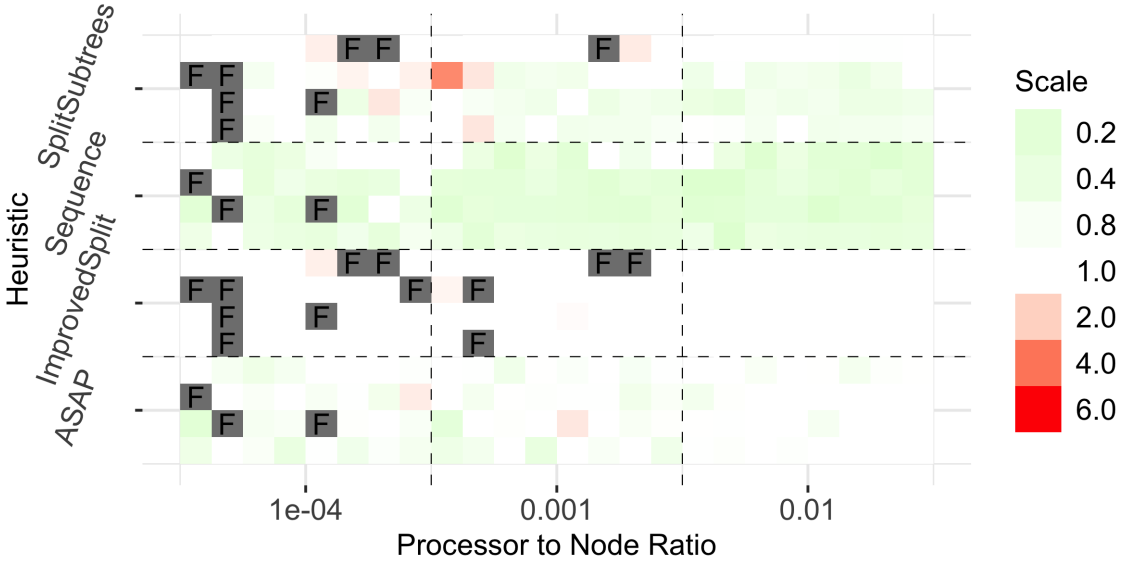
Figure 2.7 shows the performance of SplitAgain or Merge. We compare the makespan after Step 3 to the execution time that was achieved at the end of Step 2 (with an infinite number of processors), using LargestFirst, FirstFit, or Immediately during Step 2 (*strict* memory scenario), with CCR=0.1. Each tile in the figure represents a testing case, and F represents a failure, i.e., we were not able to obtain a solution with less subtrees than processors. As expected, the less processors, the more failures we have. Since Sequence did nothing in Step 1, it starts from a sequential execution of the tree and hence it obtains important gains in makespan after using SplitAgain in Step 3. SplitAgain also allows us to improve the makespan with ASAP and SplitSubtrees. As noted before, ImprovedSplit usually generates more subtrees than processors after Step 2, and hence we must use Merge to obtain a feasible solution, as well as for other heuristics when there are few processors (PNR=1e-04). We observe some failures in these cases, in particular when using ImprovedSplit, while ASAP and Sequence succeed in most cases.

Overall, the failure rate after Step 3 is 7.26% when using LargestFirst at Step 2. Compared to using LargestFirst, using FirstFit has less cases who has an increase in makespan (i.e., less red rectangles). The corresponding failure rate is 7.26%, the same as using LargestFirst. Using Immediately at Step 2, as shown in Figure 2.7c, there are more failures than using LargestFirst or FirstFit. Even using Sequence at Step 1 and PNR= 0.001, it may fail. Overall, the failure rate is 14.52% for using Immediately, two times larger than using LargestFirst.

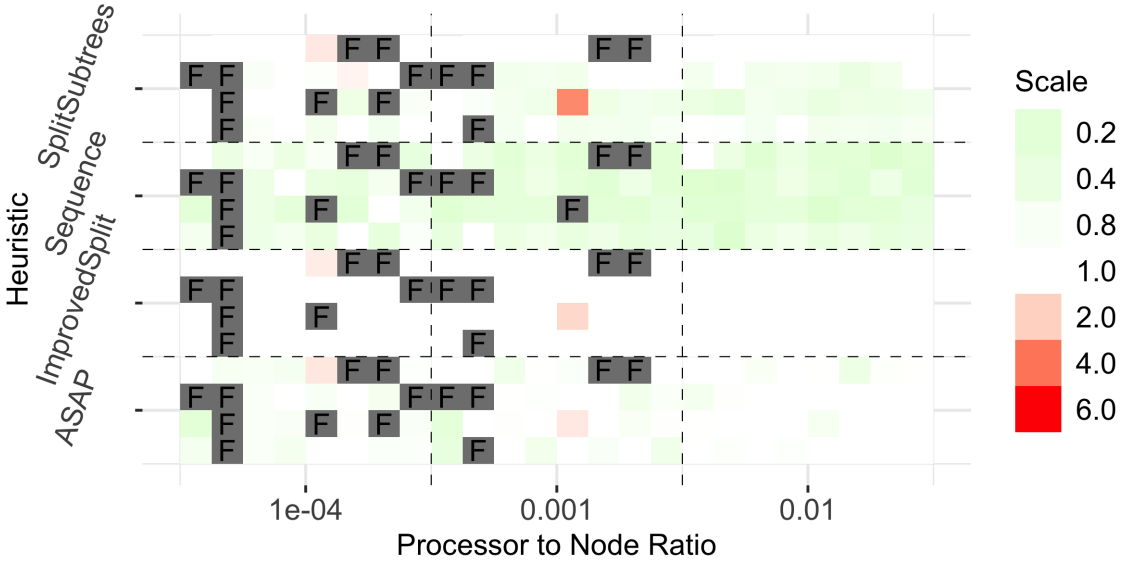
Still in the *strict* memory scenario, we finally compare the makespan of all our heuristics to FirstFit, since it is a simple adaptation from [50]. Indeed, FirstFit is likely to give a feasible solution in most cases, since it consumes least processors, as shown in Table 2.1. Furthermore, we consider the heuristic Select, which runs all possible heuristics at Step 1



(a) Use LargestFirst at Step 2

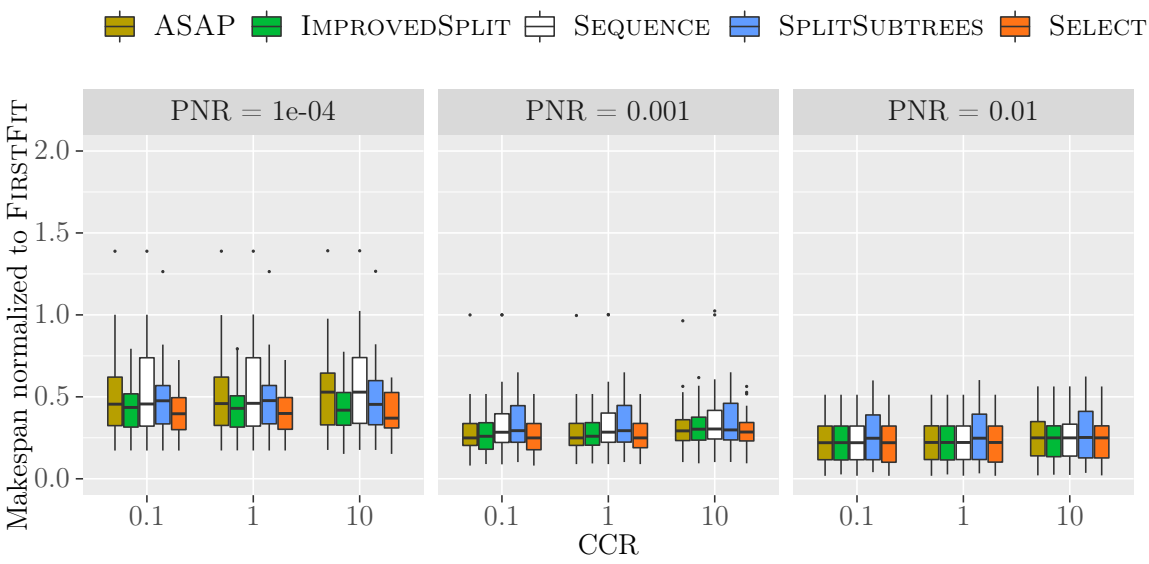


(b) Use FirstFit at Step 2

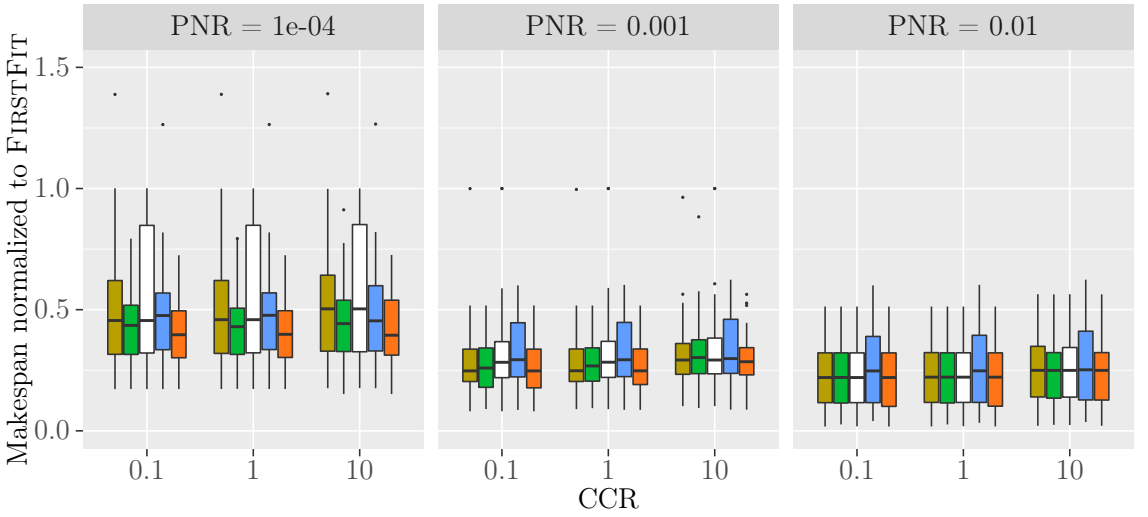


(c) Use Immediately at Step 2

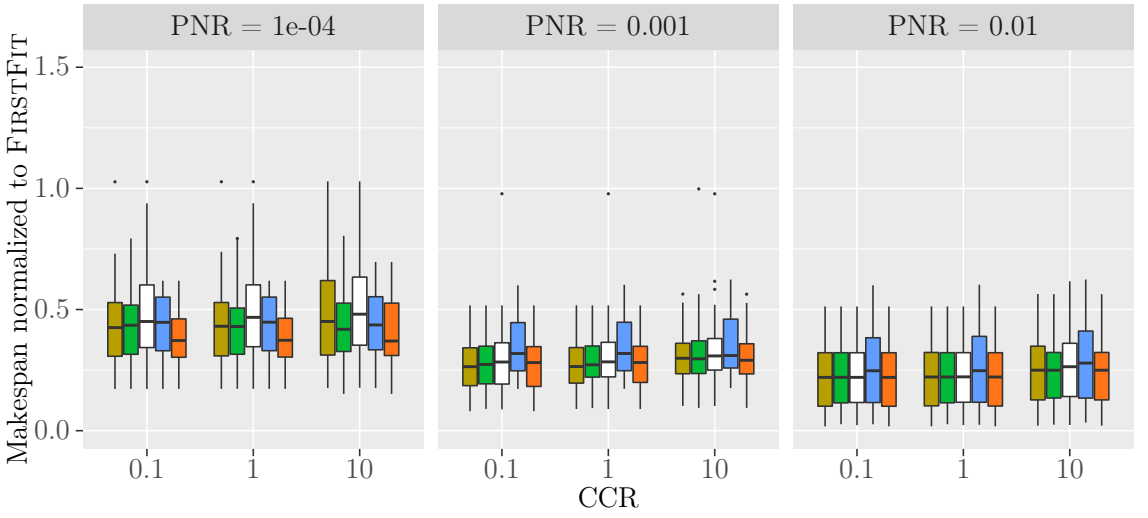
Figure 2.7 – Increase or decrease of makespan after Step 3. F represents failures. $CCR = 0.1$.



(a) use LargestFirst at step 2.



(b) use FirstFit at step 2.



(c) use Immediately at step 2.

Figure 2.8 – Final makespan (using Step 1 heuristic followed by a Setp 2 heuristic and SplitAgain/Merge) normalized to FirstFit.

(followed by a heuristic of Step 2 (LargestFirst, FirstFit or Immediately), and then SplitAgain or Merge), and keeps the best solution for each input tree. This allows us to analyze whether there is at least one heuristic that outperforms others in all situations.

Figure 2.8 presents the final makespan obtained after all three steps, excluding cases on which no solution was found. Recall that failure rates can be found in Figure 2.7. We first illustrate the results with using LargestFirst at step 2. Overall, ImprovedSplit is the best heuristic when there are a few processors (PNR=1e-04), but other heuristics outperform it in several cases, since Select is even better achieving a makespan 2.5 times faster than the reference FirstFit. With more processors, ASAP is slightly better, in particular for PNR=0.001. Skipping Step 1 (Sequence) gives reasonable results as soon as there are many processors (PNR=0.01, or even PNR=0.001), which shows that the heuristics from Step 3 (SplitAgain and Merge) are very efficient in these cases (in particular SplitAgain). With PNR=0.01, all heuristics achieve a makespan four times smaller than the reference, in average. Note that we may get a makespan worse than the reference on some cases, in particular with Sequence and SplitSubtrees (1.39 or 1.26 times worse than the makespan from FirstFit), but these outlier cases are avoided by using Select.

When using FirstFit at step 2, the biggest difference is that the makespan of Sequence (followed by SplitAgain/Merge in step 3) is worse than using LargestFirst at step 2, when there are a few processors (PNR=$1e-04$). When using Immediately at step 2, results are slightly different than with LargestFirst, and overall, ASAP becomes the best solution, then followed by ImprovedSplit and Sequence. When there are a few processors (PNR= $1e-04$), no heuristic always win over the others, since the median of Select is around 0.08 lower than the others. Excluding failure cases, the final makespan using Immediately at Step 2 is slightly better than when using LargestFirst. This is more obvious when there are few processors (PNR= $1e-04$): compared to using LargestFirst, ASAP decreases 8% in average when using Immediately.

Of course, Select is always the best pick, but it may come at the price of a higher scheduling time, since it implies to run all four variants. We report the execution times in Figure 2.9. To ease the reading, we plot the scheduling time (in minutes) and number of nodes in the tree on logarithmic scale axes. We first discuss the result with using LargestFirst at step 2. ASAP and Sequence are the fastest heuristics, and it is interesting to note that ASAP can sometimes be even faster than Sequence, even though Sequence does not do anything in Step 1: the tree obtained at the end of Step 1 with ASAP has then a faster scheduling time for Steps 2 and 3 than starting from the original tree. As expected, ImprovedSplit takes more time than SplitSubtrees, since it refines the solution from SplitSubtrees to cut in several levels. It has to be noted that very long scheduling times (above 10 minutes) only happen for very large trees (above 100.000 nodes), except for a few extreme cases. Overall, running Select is only slightly longer than ImprovedSplit, since the scheduling times of all other heuristics are small in comparison to the one of ImprovedSplit. The same conclusion holds for the case when using FirstFit at step 2. When using Immediately at step 2, the only difference to using LargestFirst is that Sequence becomes the fastest one when PNR= 0.01. Finally, note that different processor to node ratios (PNR) only slightly impact the scheduling time (see Figure 2.10 for detailed results).

To summarize, we recommend using Select with LargestFirst at step 2, unless the scheduling time is very important or the tree is very large, in which cases ASAP is a good option for Step 1 (efficient makespan obtained with a fast execution of the heuristic).

Finally, we present results in the *loose* memory scenario, where the memory bound for each processor is set to *MinMemory*, hence there is no memory constraint. In this case, we only consider the use of Step 1 directly followed by Step 3. The reference heuristic becomes SplitSubtrees, which was directly adapted from ideas from [38], resulting in a two-level split of the tree.

Figure 2.11 reports the final makespan, after applying one heuristic of Step 1 followed by

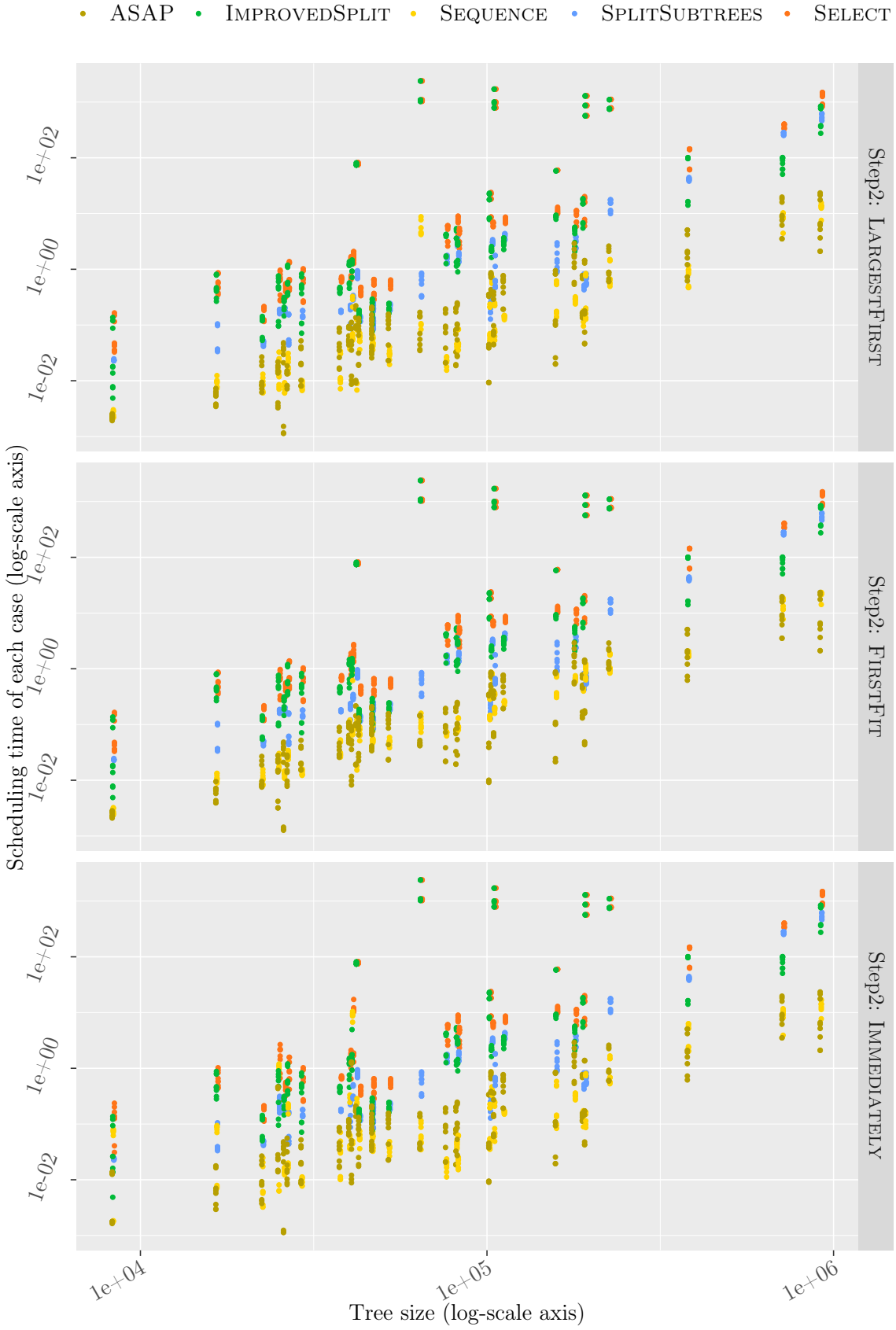


Figure 2.9 – Logarithmic scheduling time in minutes of different allocation policies, all followed by SplitAgain or Merge.

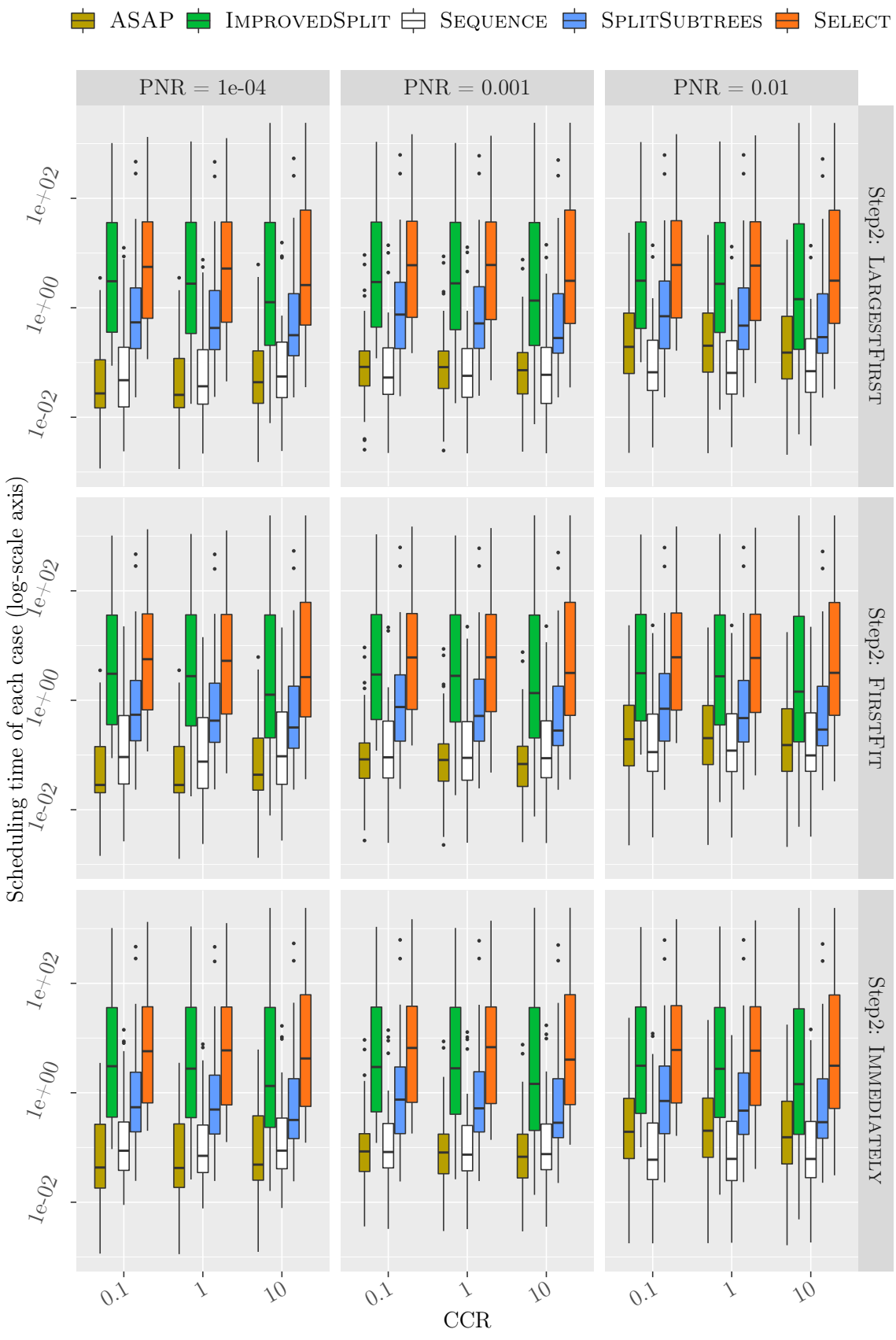


Figure 2.10 – Logarithmic scheduling time in minutes of different allocation policies, all followed by SplitAgain or Merge.

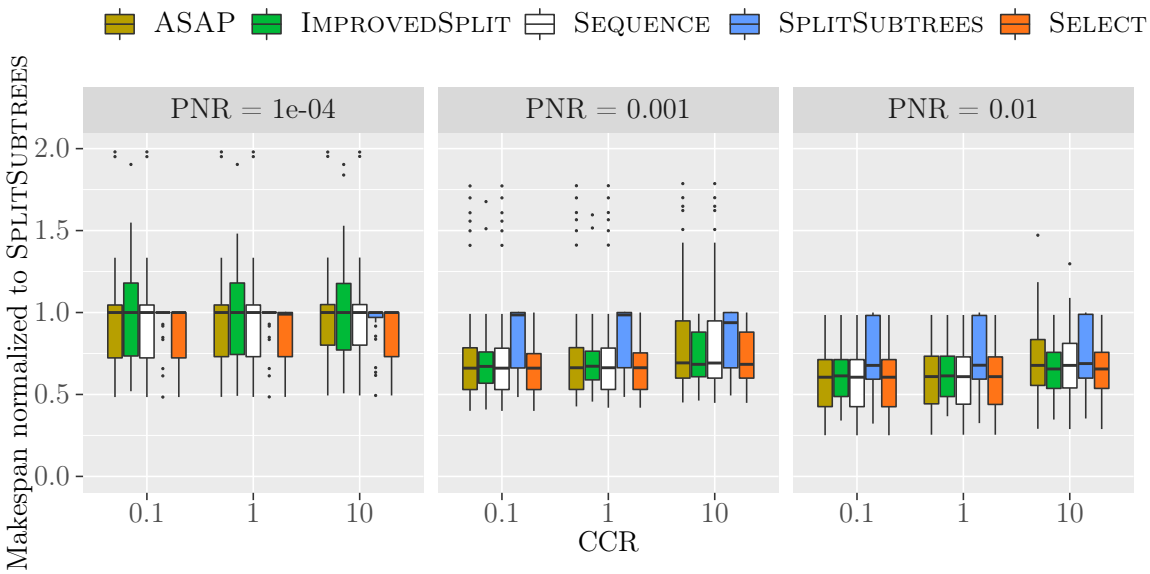


Figure 2.11 – Final makespan of each Step 1 heuristic followed by SplitAgain, normalized to SplitSubtrees in the *loose* memory scenario.

SplitAgain. Indeed, at the end of Step 1, there are always less subtrees than processors. With few processors (PNR=1e-04), there is little room for improvement over the two-level partitioning of the tree. However, when the number of processors increase, SplitAgain achieves good results in using available processors to reduce the makespan, even without going through a heuristic from Step 1 (Sequence variant). Using only SplitAgain on the original tree is a good option in this case.

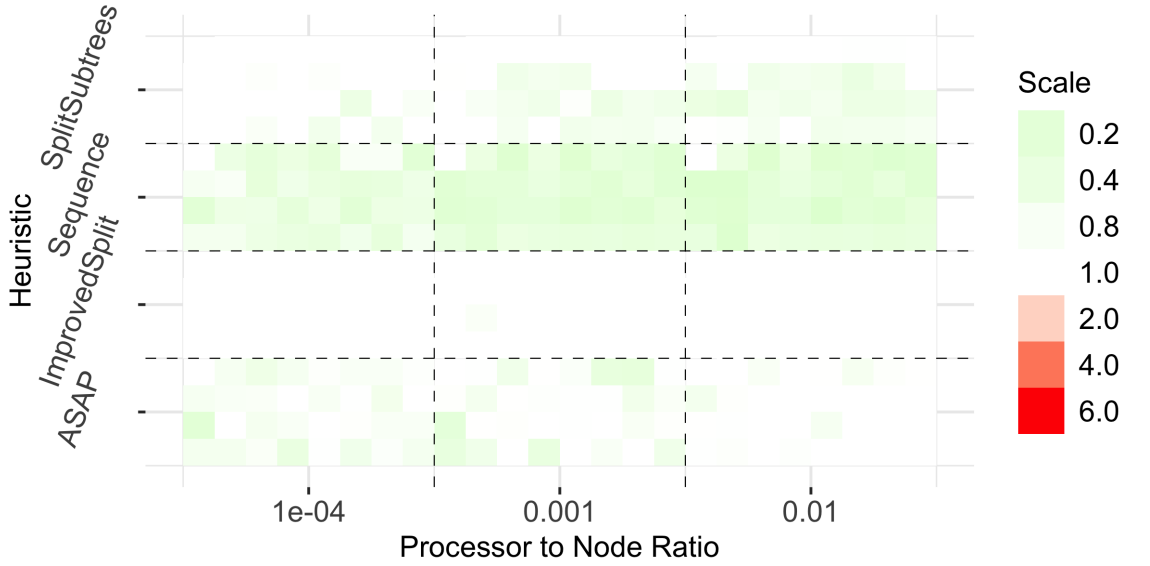


Figure 2.12 – Increase or decrease of makespan after Step 3, in the *loose* memory scenario. CCR= 0.1.

Figure 2.12 shows the performance of SplitAgain, after heuristics of Step 1, no heuristics of Step 2 is used since the local memory is set as *MinMemory*. Obviously, there is no failure case. Indeed, heuristics of Step 1 guarantee that no more subtrees are generated than processors. Makespan is decreased by SplitAgain on most cases except ImprovedSplit, as it always takes fully advantage of processors, leaves no idle processors for improvement.

Finally, Figure 2.13 reports the execution times of a heuristic of Step 1 followed by SplitAgain. Sequence is the fastest one, then closely followed by ASAP. Improved-

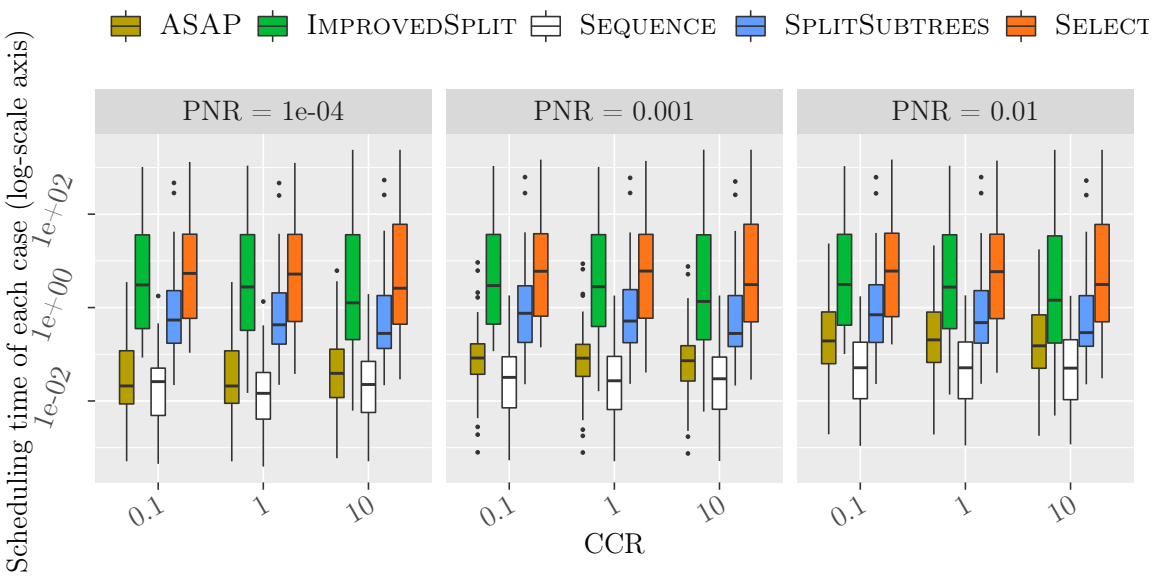


Figure 2.13 – Logarithmic scheduling time in minutes of different allocation policies, all followed by SplitAgain, in the *loose* memory scenario.

Split still takes far more time than SplitSubtrees. Select is only slightly longer than ImprovedSplit.

2.7 Chapter summary

We have studied a tree partitioning problem, targeting at a multiprocessor computing system in which each processor has its own local memory. The tree represents dependencies between tasks, and it can be partitioned into subtrees, where each subtree is executed by a distinct processor. The goal is to minimize the time required to compute the whole tree (makespan), given some memory constraints: the minimum memory requirement of traversing each subtree should not be more than the local memory capacity. We have proved that the problem above, MinMakespan, is NP-complete, and we have designed several heuristics to tackle it. We propose a three-step approach: (i) minimize the makespan; (ii) fit into memory if needed; and (iii) make sure that we have less subtrees than processors, and use as many processors as required to further minimize the makespan.

Extensive simulations demonstrate the efficiency of these heuristics and provide guidelines about the heuristics that should be used. Without memory constraint, the heuristic from Step 3, SplitAgain, is efficiently splitting the tree to minimize the makespan, and achieves results 1.5 times better than the reference heuristic, SplitSubtrees, when there are many processors available (processor to node ratio PNR≥ 0.001). When there are memory constraints, one must make sure that each subtree fits into memory, and the reference heuristic is FirstFit, which partitions the tree for memory. In this case, using the best combination of a heuristic of Step 1, a heuristic to fit into memory, and finally SplitSubtrees or Merge, allows us to drastically improve the makespan (two to four times better, depending on the processor to node ratio). The use of ASAP in Step 1 may be selected for a smaller scheduling time, since ImprovedSplit may lead to a smaller makespan, but at the price of a longer scheduling time.

Building upon these promising results, an interesting direction for future work would be to consider partitions that do not necessarily rely on subtrees, but where a single processor may handle several subtrees. Also, this work can be extended to general directed acyclic graphs of task, while we have restricted the approach to trees so far.

Chapter 3

Improving mapping for sparse direct solvers: A trade-off between data locality and load balancing

This work is the result of a collaboration with Grégoire Pichon, Mathieu Faverge, and Pierre Ramet from INRIA Bordeaux to solve a task tree mapping problem, arising in a parallel sparse matrix solver – PaStiX. The numerical factorization can be described as a traversal of computational tree. One of the pre-processing phases is to map the nodes onto the processors of the target architecture. In this phase, two scheduling algorithms have reverse strengths and drawbacks. The first one, named PropMap, shows a good data-locality but a bad load balance (some processors are idle sometimes). The second one, All2All, takes fully advantage of processors but induces a large amount of data movement and a high time complexity. Our goal is to propose a solution that reduces the idle time and preserves a good data-locality and a low complexity.

From the perspective of scheduling, this work is about mapping a tree of tasks onto processors in a shared memory setting and/or a distributed setting. In Chapter 2, the tree is partitioned into many small-sized subtrees such as to fit the local memory capacity, and we restrict a processor to handle only a subtree. In this work, this limit is released, a processor can handle many subtrees. Data movements between two subtrees can be saved if they are mapped onto the same processor. The goal in this chapter is hence to reduce the execution time by taking fully advantage of parallel platforms (i.e., reduce idle time of processors) and reducing the data movement between processors, since it is time expensive, especially in a distributed setting. From the practical point of view, this work could be seen as an application of tree scheduling in a real parallel software. The role of task trees and how do we estimate the execution time of the solver by simulations of processing the tree on a parallel platform are presented in detail. This work has been published at Euro-Par 2020 [C2].

3.1 Introduction

For the solution of large sparse linear systems, we design numerical schemes and software packages for direct parallel solvers. Sparse direct solvers are mandatory when the linear system is very ill-conditioned for example [29]. Therefore, to obtain an industrial software tool that must be robust and versatile, high-performance sparse direct solvers are mandatory, and parallelism is then necessary for reasons of memory capability and acceptable solution time. Moreover, in order to solve efficiently 3D problems with several million unknowns, which is now a reachable challenge with modern supercomputers, we must achieve good scalability in time and control memory overhead. Solving a sparse linear system by a direct method is generally a highly irregular problem that provides some challenging algorithmic problems and requires a sophisticated implementation scheme in order to fully exploit the

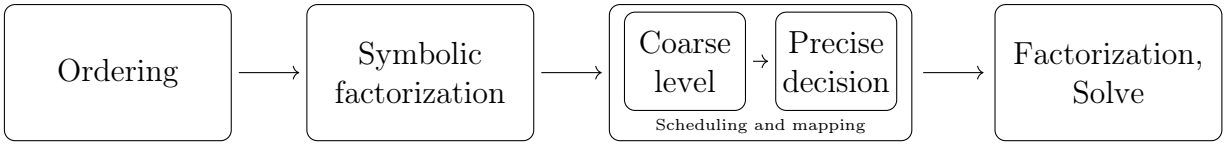


Figure 3.1 – A workflow composed of sequential preprocessing steps, parallel factorization and solve steps in PaStiX.

capabilities of modern supercomputers.

There are two main approaches in direct solvers: the multifrontal approach [5, 36], and the supernodal one [46, 74]. Both can be described by a computational tree whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, at each node, some steps of Gaussian elimination are performed on a dense frontal matrix and the remaining Schur complement, or contribution block, is passed to the parent node for assembly. In the case of the supernodal method, the distributed memory version uses a right-looking formulation which, having computed the factorization of a supernode corresponding to a node of the tree, then immediately sends the data to update the supernodes corresponding to ancestors in the tree. In a parallel context, we can locally aggregate contributions to the same block before sending the contributions. This can significantly reduce the number of messages. Independently of these different methods, a static or dynamic scheduling of block computations can be used. For homogeneous parallel architectures, it is useful to find an efficient static scheduling.

In order to achieve efficient parallel sparse factorization, as shown in figure 3.1, we perform the three sequential preprocessing phases:

1. The ordering step, which computes a symmetric permutation of the initial matrix such that the factorization process will exhibit as much concurrency as possible while incurring low fill-in.
2. The block symbolic factorization step, which determines the block data structure of the factorized matrix associated with the partition resulting from the ordering phase. From this block structure, one can deduce the weighted elimination quotient graph that describes all dependencies between column-blocks, as well as the supernodal elimination tree.
3. The block scheduling/mapping step, which consists in mapping the resulting blocks onto the processors according to a 1D scheme (i.e. by supernode) for the lower part of the tree, and according to a 2D scheme (i.e. by blocks) for the upper part of the tree. During this mapping phase, a static optimized scheduling of the computational and communication tasks, according to models calibrated for the target machine, can be computed.

When these preprocessing phases are done, the computation on the actual data, that is the numerical factorization, can start parallelly.

The optimization problem that needs to be solved at the scheduling/mapping stage is known to be NP-hard, and is usually solved using a proportional mapping heuristic [70]. This mono-constraint heuristic induces idle times during the numerical factorization. In this chapter, we extend the proportional mapping and scheduling heuristic to reduce these idle times. We first detail in Section 3.2 proportional mapping heuristic with its issues and related work, before describing the original application in the context of the PaStiX solver [47] in Section 3.3. Then, in Section 3.4, we explain the introduced solution before studying its impact on a large set of test cases in Section 3.5. Conclusions are presented in Section 3.6.

3.2 Related work

Among different mapping strategies that are used by both supernodal and multifrontal sparse direct solvers, the subtree to subcube mapping [44] and the proportional mapping [70] are the most popular. These approaches consist of tree partitioning techniques, where the set of resources mapped on a node of the tree are split among disjoint subsets, each mapped to a child subtree.

The proportional mapping method performs a top-down traversal of the elimination tree, during which each node is assigned a set of computational resources. All the resources are assigned to the root node, which performs the last task. Then, the resources are split recursively following a balancing criterion. The set of resources dedicated to a node are split among its children, proportionally to their weight or any other balancing criterion. This recursive process ends at the leaves of the tree, or when entire subtrees are mapped onto a single resource.

The original version of the proportional mapping [70] computes the splitting of resources depending on the workload of each subtree, but more sophisticated metrics can also be used. In [71], a scheduling strategy was proposed for tree-shaped task graphs. The time for computing a parallel task (for instance at the root node of the elimination tree) is considered as proportional to the length of the task and to a given parallel efficiency. This method was proven efficient in [12] for a multifrontal solver. The proportional mapping technique is widely used because it helps reducing the volume of data transfers due to its data locality. In addition, it allows us to exhibit both tree and node parallelism.

Note that alternative solutions to the proportional mapping have been proposed, such as the 2D block-cyclic distribution of SuperLU [60], or the 1D cyclic distribution of symPACK [51]. In the latter, the non load-balanced solution is compensated by a complex and advanced communication scheme that balances the computations in the nodes to get good performance results out of this mapping strategy.

As stated earlier, sparse direct solvers commonly use the proportional mapping heuristic to distribute supernodes (a full set of columns, i.e., 1D distribution that share the same row pattern) onto the processors. Note that each supernode can be split into smaller nodes to increase the level of parallelism, which modifies the original supernodes tree structure as shown in Figure 3.2. This heuristic provides a set of candidate processors for each supernode, which is then refined dynamically when going up the tree, as in MUMPS [4] or PaStiX [47], with a simulation stage that affects a single processor among the candidates, while providing a static optimized scheduling. The proportional mapping stage, by its construction, may however introduce idle time in the scheduling. This is illustrated on Figure 3.3. The ten candidate processors of the root node are distributed among the two sons of weight respectively 4 and 6. The Gantt diagram points out the issue of considering a single criterion heuristic to set the mapping: no work is given to processor p_9 due to the low level of parallelism of the right node, whereas it could benefit to the left node.

3.3 Description of the application

At a coarse-grain level, the computation can be viewed as a tree T whose nodes (or vertices) represent supernodes of the matrix, and where the dependencies are directed towards the root of the tree. A supernode is a set of contiguous columns with the same row pattern. Because sparse matrices usually represent physical constraints and thanks to the nested dissection used to order the matrix, nodes at the bottom of the tree are usually small and nodes at the top are much larger. Each supernode is itself a small DAG (Directed Acyclic Graph) of tasks as illustrated on Fig. 3.2. A more refined view shows that the dependencies between two supernodes consist of dependencies between tasks of these supernodes. Another way to put it is that the computation is described as a DAG of tasks, tasks are partitioned into supernodes, and the quotient graph of supernodes is the tree T (with some transitivity

edges). Note that with 1D distribution, as targetted here, the DAG within can also be seen as a tree with dependencies toward the roots. Thus, in this study, we will use either nodes or supernodes to denote the vertices of the tree T as they can be used interchangeably.

This structure in two levels allows us to both reduce the cost of the analysis stage by considering only the first level, while increasing the parallelism level during the numerical factorization with finer grain computations.

We denote by $root(T)$ the node at the root of tree T, and by w_i the computational weight of the node i, for $1 \leq i \leq n$: this is the total number of operations of all tasks within node i. Also, $parent(i)$ is the parent of node i in the tree (except for the root), and $child(i)$ are the children nodes of i in the tree. Given a subtree T_i of T (rooted in $root(T_i)$), $W_i = \sum_{j \in T_i} w_j$ is the computational weight of this subtree.

As stated above, each node i of the tree is itself made of $n_i \geq 1$ tasks $i_1, \ldots, i_{n_i}$, whose dependencies follow a directed acyclic graph (DAG). Each of these tasks is a linear algebra kernel (such as matrix factorization, triangular solve or matrix product) on block matrices. Hence, given a node i and its parent $j = parent(i)$ in the tree, only some of the tasks of i need to be completed before j is started, which allows some pipelining in the processing of the tree.

When running on a parallel platform with a set P of p processors, nodes and tasks are distributed among available processing resources (processors) in order to ensure a good load-balancing. If node i is executed on $alloc[i] = k$ processors, its execution time is $f_i(k)$; this time depends on w_i and on the structure of the DAG of tasks.

Following the structure of the application, the mapping is done in two phases: the first phase, detailed in Section 3.3.1, consists in using the Proportional Mapping algorithm [70] to compute a mapping of nodes to subsets of processors. The second phase, detailed in Section 3.3.2, refines this mapping by allocating each task of a node i to a single processor of the subset allocated to i in the first step.

3.3.1 Coarse-grain load balancing using proportional mapping

The proportional mapping process follows the sketch of Algorithm 6. First, all processors are allocated to the root of the tree. Then, we compute the total weight of its subtrees (i.e., the sum of the weight of their nodes), and allocate processors to subtrees so that the load is balanced. Then, we recursively apply the same procedure on each subtree.

Apart from balancing the load among branches of the tree, the proportional mapping is known for its good data locality: a processor is allocated to nodes of a single path from a leaf to the root node, and only to nodes on this path. Thus, the data produced by a node and used by its parents mostly stay on a single processor, and no data transfer is made except for the necessary redistribution of data in the upper levels of the tree. This is particularly

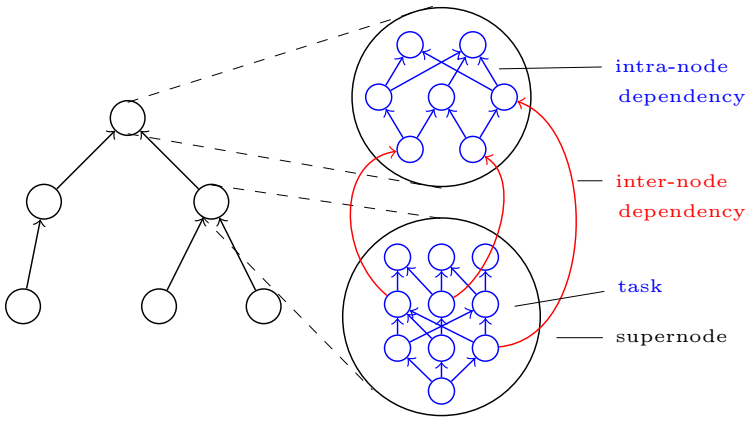


Figure 3.2 – Structure of the computation: tree of supernodes, each supernodes being made of several tasks..

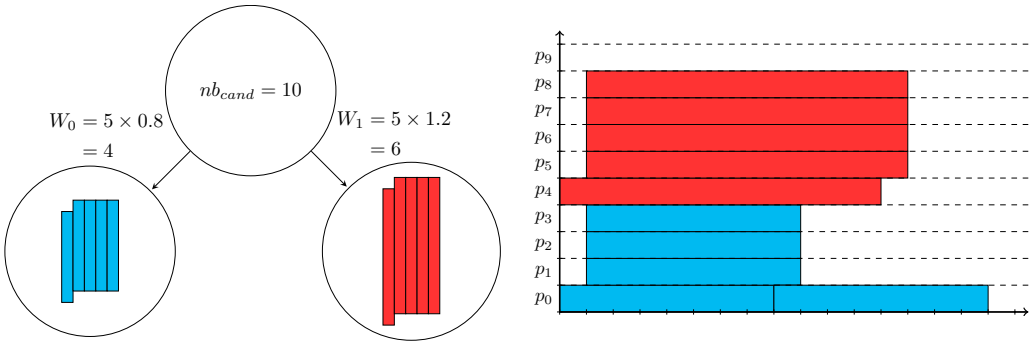


Figure 3.3 – Illustration of proportional mapping: elimination tree on the left, and Gantt diagram on the right. Based on a single weight criterion. On the left, the elimination tree presents a single node with 10 potential candidates and two sons of respective costs of 4 and 6. On the right, the associated Gantt diagram for these two nodes with the regular weight mapping.

Algorithm 6 Proportional mapping with integer number of processors

function *PropMapInt*(tree T, set P of processors):
Allocate all processors in P to the root of tree T
For each subtree T_i of T, compute its total weight W_i
Find subsets of processors P_i such that $\max(W_i/|P_i|)$ is minimal and $\sum |P_i| = |P|$
For each subtree T_i of T, call *PropMapInt*(T_i, P_i)

interesting in a distributed context, where communications among processors are costly.

However, the mapping algorithm 7 suffers a major problem when used in a practical context, because they forbid allocating processors to more than one child of a node. First, some nodes, especially leaves, have very small weight and several of them should be mapped on the same processor. Second, allocating integer numbers of processors to nodes creates unbalanced workloads, for example, when three processors have to be allocated to two identical subtrees. All implementations of the proportional mapping tackle this problem (including the first one in [70]). For example, the actual implementation in PaStiX, as sketched in Algorithm 7, allows "border processors" to be shared among branches, and keeps track of the occupation of each processor to ensure load-balancing. It first computes the total time needed to process the whole tree, and sets the initial availability time of each processor to an equal share of this total time. Whenever some (fraction of a) node is allocated to a processor, its availability time is reduced. Hence, if a processor is shared on two subtrees T_1, T_2, the work allocated by T_1 is taken into account when allocating resources for T_2. Note also that during the recursive allocation process, the subtrees are sorted by non-increasing total weights before being mapped to processors. This allows us to group small subtrees together in order to map them on a single processor, and to avoid unnecessary splitting of processors.

3.3.2 Mapping refinement after the coarse-grain mapping

After allocating nodes of the tree to subsets of processors, a precise mapping of each task to a processor has to be computed. In PaStiX, this is done by simulating the actual factorization, based on the prediction of both the running times of tasks and of the time needed for data transfers. The refined mapping process is detailed in Algorithm 8. Thanks to the previous phase, we know that each task can run on a subset of processors (the subset associated to the node it belongs to), called *candidate processors* for this task. We associate to each processor a ready queue, containing tasks whose predecessors have already completed, and a waiting queue, with tasks that still have some unfinished predecessor. At

Algorithm 7 Proportional mapping with shared processors among subtrees

function *ProportionalMappingShared*(tree T, number of processors p):
for each processor $k = 1, \ldots, p$ **do**
  $avail\_time[k] = \sum_{i \in T} w_i / p$
end for
Call *PropMapSharedRec*$(T, 1, p)$

function *PropMapSharedRec*(subtree T, indices *first_proc*, *last_proc*):
if *last_proc* = *first_proc* **then**
  Map all nodes in subtree T to processor *first_proc*
  $avail\_time[first\_proc] = avail\_time[first\_proc] - \sum_{i \in T} w_i$
else
  Map node $r = root(T)$ to all processors in *first_proc*, ..., *last_proc*
  for each $k =$ *first_proc*, ..., *last_proc* **do**
    $avail\_time[k] = avail\_time[k] - w_r/(last\_proc - first\_proc)$
  end for
  $next\_proc \leftarrow first\_proc$
  Sort the subtrees of T by non-increasing total weight
  for each subtree T_i in this order **do**
    $cumul\_time \leftarrow 0$
    $w_{subtree} \leftarrow \sum_{j \in T_i} w_j$
    $first\_proc\_for\_subtree \leftarrow next\_proc$
    while $cumul\_time < w_{subtree}$ **do**
      $new\_time\_share \leftarrow min(w_{subtree} - cumul\_time, avail\_time[next\_proc])$
      $cumul\_time \leftarrow cumul\_time + new\_time\_share$
      $avail\_time[next\_proc] \leftarrow avail\_time[next\_proc] - new\_time\_share$
      if $avail\_time[next\_proc] = 0$ **then** $next\_proc \leftarrow next\_proc + 1$
    end while
    PropMapSharedRec$(T_i, first\_proc\_for\_subtree, next\_proc)$
  end for
end if

the beginning of the simulation, each task is put in the waiting queue of all its candidate processors (except tasks without predecessors, which are put in the ready queue of their candidate processors). Queues are sorted by decreasing depth of the tasks in the graph (tasks without predecessors are ordered first). The depth considered here is an estimation of the critical path length from the task to the root of the tree T.

A ready time is associated both to tasks and processors:

- The ready time $RP[k]$ of processor k is the completion time of the current task being processed by k (initialized with 0).
- The ready time $RT[i]$ of task i is the earliest time when i can be started, given its input dependencies. This is at least equal to the completion time of each of its predecessors, but also takes into account the time needed for data movement, in case a predecessor of i is not mapped on the same processor as i. The ready time of tasks with non-started predecessor is set to $+\infty$.

3.3.3 Discussion on the choice of the mapping algorithm

Since the precise mapping of each task is only decided during the mapping refinement phase, one can use a different mapping strategy to compute the set of candidate. One extreme choice consists in putting all processors in the set of candidates of all tasks. This method, called *All to All mapping*, has two drawbacks:

- The time needed for the simulation phase can be very long when the number of

Algorithm 8 Precise scheduling and mapping using simulation

for all task i **do**
If i is a leaf, put i in the ready queue of every processor in $candidate(i)$, otherwise put it in the waiting queue.
end for
while all tasks have not been mapped **do**
For each processor k, consider the triplet $\langle i, k, t\rangle$ where i is the first task in the ready queue of processor k and t is the starting time of i on k ($t = \max(RT[i], RP[k])$)
Consider F, the set of all such triplets
Select the triplet $\langle i, k, t\rangle$ in F with the smallest t (if ties, choose the one with largest depth)
Schedule task i on processor k at time t
Update the ready times of processor k and of the successors of i on all their candidate processors
Update the ready queue and waiting queue of processor k, as well as of candidates processors of successors of i
end while

processors and tasks are large, since its complexity is $O(npC)$, where C is the size of the largest candidate set.

- The data locality of the obtained allocation might be very bad: tasks from the same supernode may be mapped on remote processors, inducing a large amount of data movement and contention on communications.

The task allocation based on the proportional mapping algorithm does not have these drawbacks: the set of candidate per task is limited, and the obtained data locality is very good, as discussed above. However, its hard constraint on candidate set sometimes creates load unbalance: because of the limited parallelism in some supernodes, some processors from its allocated subset may be idle. This results in a longer time needed to process all tasks of this supernodes, which in turn causes idle time in other branches, when the processing of its siblings have completed but the parent supernode cannot start being processed.

3.4 Proposed mapping refinement

Our objective is to correct the potential load imbalance (and thus idle times) created by the proportional mapping, as outlined in Section 3.2, but without impacting too much the data locality. We propose a heuristic based on work stealing [16] that extends the refined mapping phase (see Algorithm 8) using simulation (see Algorithm 9). Intuitively, we propose that if the simulation predicts that a processor will be idle, this processor tries to steal some tasks from its neighbors.

In the proposed refinement, we replace the update of the ready and waiting queues of the last line in Algorithm 8 by a call to *UpdateQueuesWithStealing* (Algorithm 9). For each processor k, we first detect if k will have some idle time, and we compute the duration d of this idle slot. This happens in particular when the ready time of the first task in its waiting queue is strictly larger than the ready time of the processor ($RT[i] > RP[k]$) and ready queue is empty. Whenever both queues are empty, the processor will be idle forever, and thus d is set to a large value. Then, if an idle time is detected (the ready queue is empty and d is a positive value), a task is stolen from a neighbor processor using function *StealTask*. Otherwise, the ready and waiting queues are updated as previously: the tasks of the waiting queue that will be freed before the processor becomes available are moved to the ready queue.

Algorithm 9 Update ready and waiting queues with task stealing

function *UpdateQueuesWithStealing*(nb. of proc. p, switch *IsSharedMem*):
for $k = 1$ **to** p **do**
  if $waiting\_queue_k \neq \emptyset$ **then**
    Let i be the first task in $waiting\_queue_k$
    $d \leftarrow RT[i] - RP[k]$
  else
    $d \leftarrow +\infty$
  end if
  if $ready\_queue_k = \emptyset$ and $d > 0$ **then**
    $StealTask(k, p, d, IsSharedMem)$
  else
    Let i be the first task in $waiting\_queue_k$
    while $RT[i] \leq RP[k]$ **do**
      Move task i from $waiting\_queue_k$ to $ready\_queue_k$
      Let i be the first task in $waiting\_queue_k$
    end while
  end if
end for

function *StealTask*(proc. k, proc. nb. p, idle time d, switch *IsSharedMem*):
if *IsSharedMem* = *false* **then**
  set $S_k \leftarrow \{k-1, k+1, k-2, k+2\}$; set $S \leftarrow \emptyset$
  for $j = 1$ **to** 4 **do**
    if $S_k[j] \geq 0$, $S_k[j] < p$, $S_k[j]$ is in the same cluster as k and $|S| < 3$ **then**
      add $S_k[j]$ to S
    end if
  end for
end if
if *IsSharedMem* = *true* or S is empty **then**
  set $S \leftarrow \{k-1 \pmod p, k+1 \pmod p\}$
end if
Build the set O with the first element of each ready queue of processors in S
Let o be the task of O with minimum $RT[o]$
if $RT[o] < RP[k] + d$, **then** insert o into $ready\_queue_k$

When stealing tasks, we distinguish between two cases, depending whether we use shared or distributed memory. In shared memory, the two possible victims of the task stealing operation are the two neighbors of processor k, considering that processors are arranged in a ring. In the case of distributed memory, we first try to steal from two neighbor processors *within the same cluster*, that is, within the set of processors that share the same memory. Stealing to a distant processor is considered only when clusters are reduced to a single element. Once steal victims are identified (set S), we consider the first task of their ready queues and select the one that can start as soon as possible. If the task is able to start during the idle slot of processor k (and thus reduce its idle time), it is then copied into its ready queue.

3.5 Experimental results

Experiments were conducted on the *Plafrim*[1] supercomputer, and more precisely on the `miriel` cluster. Each node is equipped with two Intel Xeon E5-2680v3 12-cores running at 2.50 GHz and 128 GB of memory. The Intel MKL `2019` library is used for sequential BLAS kernels. Another shared memory experiment was performed on the `crunch` cluster from the LIP[2], where a node is equipped with four Intel Xeon E5-4620 8-cores running at 2.20 GHz and 378 GB of memory. On this platform, the Intel MKL `2018` library is used for sequential BLAS kernels. The PaStiX version used for our experiments is based on the public git repository[3] version at the tag `europar2020`.

In the following, the different methods used to compute the mapping are compared. All to All, referred to as All2All, and Proportional mapping, referred to as PropMap, are available in the PaStiX library, and the newly introduced method is referred to as Steal. When the option to limit stealing tasks into the same MPI is enabled, we refer to it as StealLocal. In all the following experiments, we compare these versions with respect to the All2All strategy, which provides the most flexibility to the scheduling algorithm to perform load balance, but does not consider data locality. The multi-threaded variant is referred to as SharedMem, while for the distributed settings, pMt stands for p MPI nodes with t threads each. All distributed settings fit within a single node.

In order to make a fair comparison between the methods, we use a set of 34 matrices issued from the SuiteSparse Matrix collection [28]. The matrix sizes range from $72K$ to $3M$ of unknowns. The number of floating point operations required to perform the LL^t, LDL^t, or LU factorization ranges from 111 GFlops to 356 TFlops, and the problems are issued from various application fields. Table 3.1 lists these matrices.

Communications. We first report the relative results in terms of communications among processors in different clusters (MPI nodes), which are of great importance for the distributed memory version. The number and the volume of communications normalized to All2All are depicted in Fig. 3.4. One can observe that all three strategies largely outperform the All2All heuristic, which does not take communications into account. The number of communications especially explodes with All2All as it mainly moves around leaves of the elimination tree. This creates many more communications with a small volume. This confirms the need for a proportional-mapping-based strategy to minimize the number of communications. Both numbers and volumes of communications also confirm the need for the local stealing algorithm to keep it as low as possible. Indeed, Steal generates 6.19 times more communications on average than PropMap, while StealLocal is as good as PropMap. Note the exception of the `24M1` case where Steal and StealLocal are

[1] `https://www.plafrim.fr`
[2] `http://www.ens-lyon.fr/LIP/`
[3] `https://gitlab.inria.fr/solverstack/pastix`

Kind	Matrix	Arith.	Fact.	N	NNZ_A
2d/3d	PFlow_742	d	LL^t	742 793	18 940 627
	nd24k	d	LL^t	72 000	28 715 634
	lap120	d	LL^t	1 728 000	6 868 800
	Bump_2911	d	LL^t	2 911 419	65 320 659
Computational fluid dynamics	StocF-1465	d	LL^t	1 465 137	11 235 263
	atmosmodl	d	LU	1 489 752	10 319 760
	atmosmodd	d	LU	1 270 432	8 814 880
	RM07R	d	LU	381 689	37 464 962
Dna electrophoresis	cage13	d	LU	445 315	7 479 343
Electromagnetics	dielFilterV3clx	z	LU	420 408	16 653 308
	fem_hifreq_circuit	z	LU	491 100	20 239 237
	dielFilterV2clx	z	LU	607 232	12 958 252
Magnetohydrodynamics	matr5	d	LU	485 597	24 233 141
Materials	3Dspectralwave2	z	LDL^h	292 008	7 307 376
	3Dspectralwave	z	LDL^h	680 943	30 290 827
Model reduction	boneS10	d	LL^t	914 898	28 191 660
	CurlCurl_3	d	LDL^t	1 219 574	7 382 096
	bone010	d	LL^t	986 703	36 326 514
	CurlCurl_4	d	LDL^t	2 380 515	14 448 191
Optimization	nlpkkt80	d	LDL^t	1 062 400	14 883 536
Structural	ldoor	d	LL^t	952 203	23 737 339
	inline_1	d	LL^t	503 712	18 660 027
	sparsine	d	LDL^t	50 000	1 548 988
	Flan_1565	d	LL^t	1 564 794	59 485 419
	ML_Geer	d	LU	1 504 002	110 879 972
	audikw_1	d	LL^t	943 695	39 297 771
	Fault_639	d	LL^t	638 802	14 626 683
	Hook_1498	d	LL^t	1 498 023	31 207 734
	Transport	d	LU	1 602 111	23 500 731
	Emilia_923	d	LL^t	923 136	20 964 171
	Geo_1438	d	LL^t	1 437 960	32 297 325
	Serena	d	LL^t	1 391 349	32 961 525
	Long_Coup_dt0	d	LDL^t	1 470 152	44 279 572
	Cube_Coup_dt0	d	LDL^t	2 164 760	64 685 452

Table 3.1 – Set of real-life matrices issued from The SuiteSparse Matrix Collection [28] (except matr5 and lap120), sorted by family and number of operations.

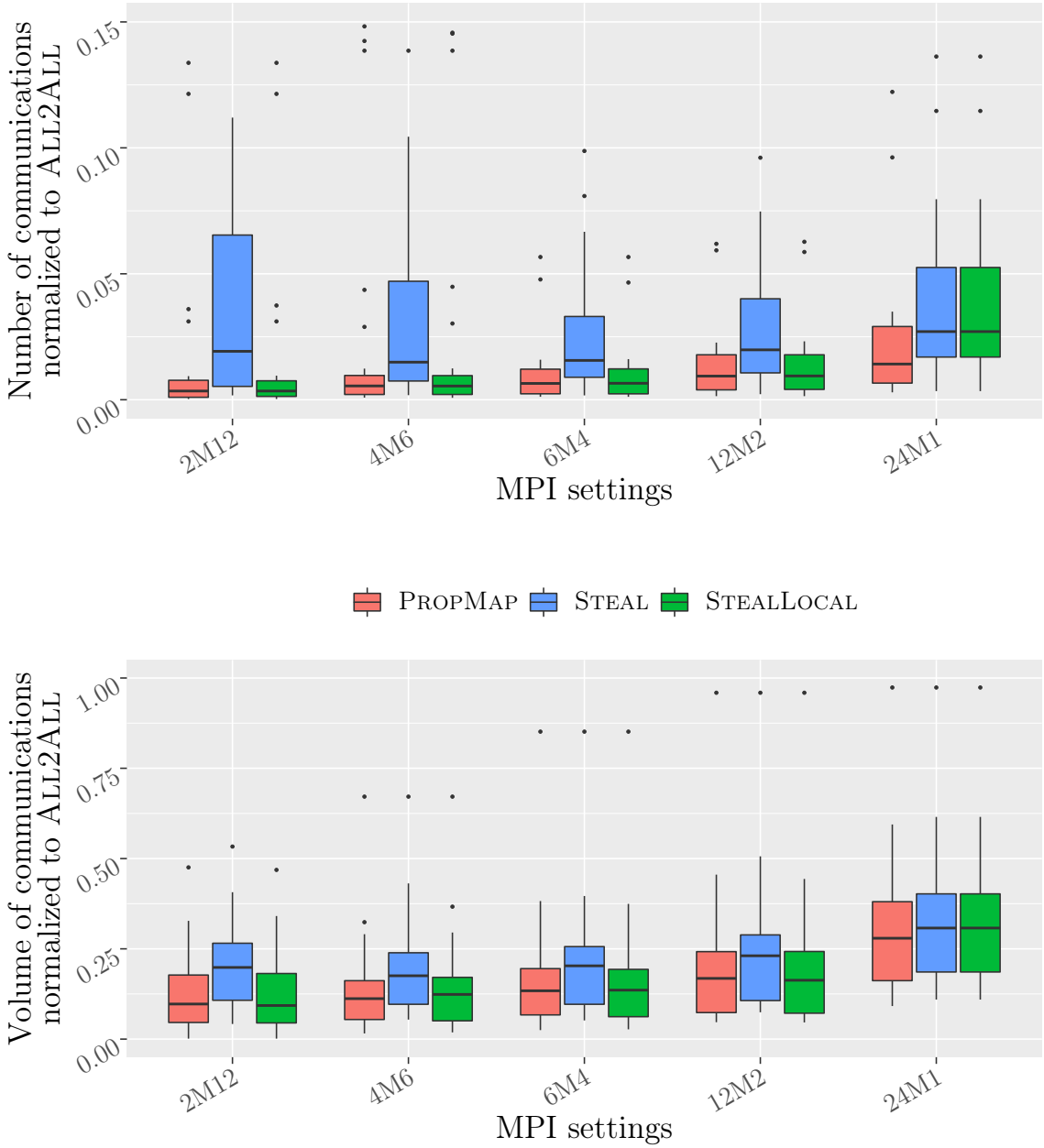


Figure 3.4 – MPI communication number (*top*) and volume (*bottom*) for the three methods: PropMap, Steal, and StealLocal, with respect to All2All.

identical. No local task can be stolen. These conclusions are similar when looking at the volume of communication with a ratio reduced to 1.92 between Steal and PropMap.

Data movements. Fig. 3.5 depicts the number and volume of data movements normalized to All2All and summed over all the MPI nodes with different MPI settings. The data movements are defined as a write operation on the remote memory region of other cores of the same MPI node. Note that accumulations in local buffers before send, also called fan-in in sparse direct solvers, are always considered as remote write. This explains why all MPI configurations have equivalent number of data movements. As expected, proportional mapping heuristics outperform All2All by a large factor on both number and volume, which can have an important impact on NUMA architectures. Compared to PropMap, Steal and StealLocal are equivalent and have respectively 1.38x, and 1.32x, larger number of data movements on average respectively, which translates into 9%, and 8% of volume increase. Note that in the shared memory case, StealLocal behaves as Steal as there is only one MPI node.

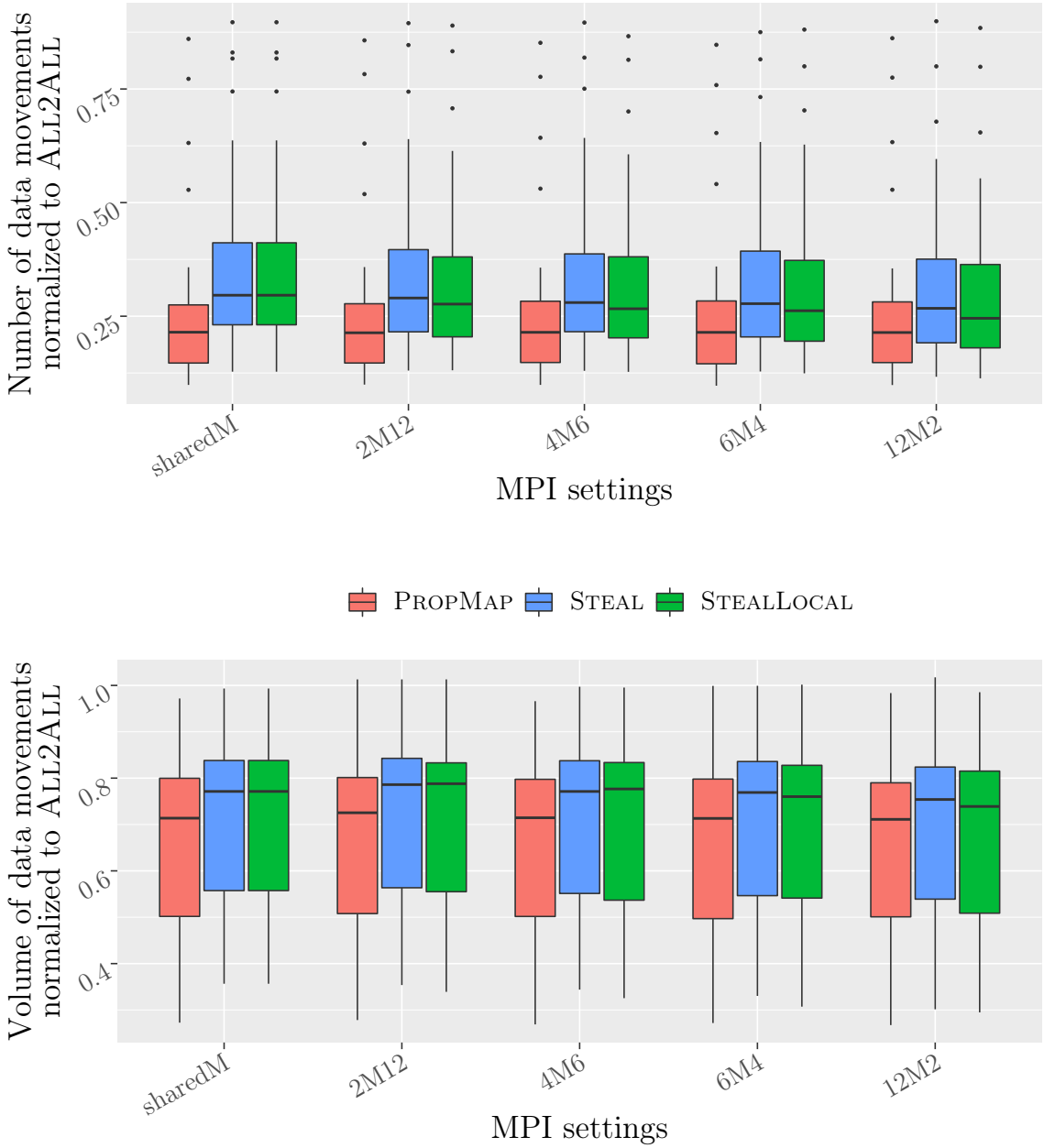


Figure 3.5 – Shared memory data movements number (*top*) and volume (*bottom*) within MPI nodes for PropMap, Steal, and StealLocal, with respect to All2All.

Simulation cost. Fig. 3.6 shows the simulation cost in seconds (duration of the refined mapping via simulation) on the top, and that of PropMap, Steal and StealLocal with respect to All2All on the bottom. Fig. 3.7 shows the original simulated factorization time obtained with these heuristics and a normalized version. Note that, for the sake of clarity, some large outliers are removed from the top subfigure of Fig. 3.7. As stated in Section 3.2, the All2All strategy allows for more flexibility in the scheduling, hence it results in a better simulated time for the factorization in average. However, its cost is already 4x larger for this relatively small number of cores. Fig. 3.6 shows that the proposed heuristics have similar simulation cost to the original PropMap, while Fig. 3.7 shows that the simulated factorization time gets closer to All2All, and can even outperform it in extreme cases. Indeed, in the `24M1` case, Steal outperforms All2All due to bad decisions taken by the latter at the beginning of the scheduling. The bad mapping of the leaves is then never recovered and induces extra communications that explain this difference. In conclusion, the proposed heuristic, StealLocal, manages to generate better schedules with a better load-balance than the original PropMap heuristic, while generating small or no overhead

on the mapping algorithm. This strategy is also able to limit the volume of communications and data movements as expected.

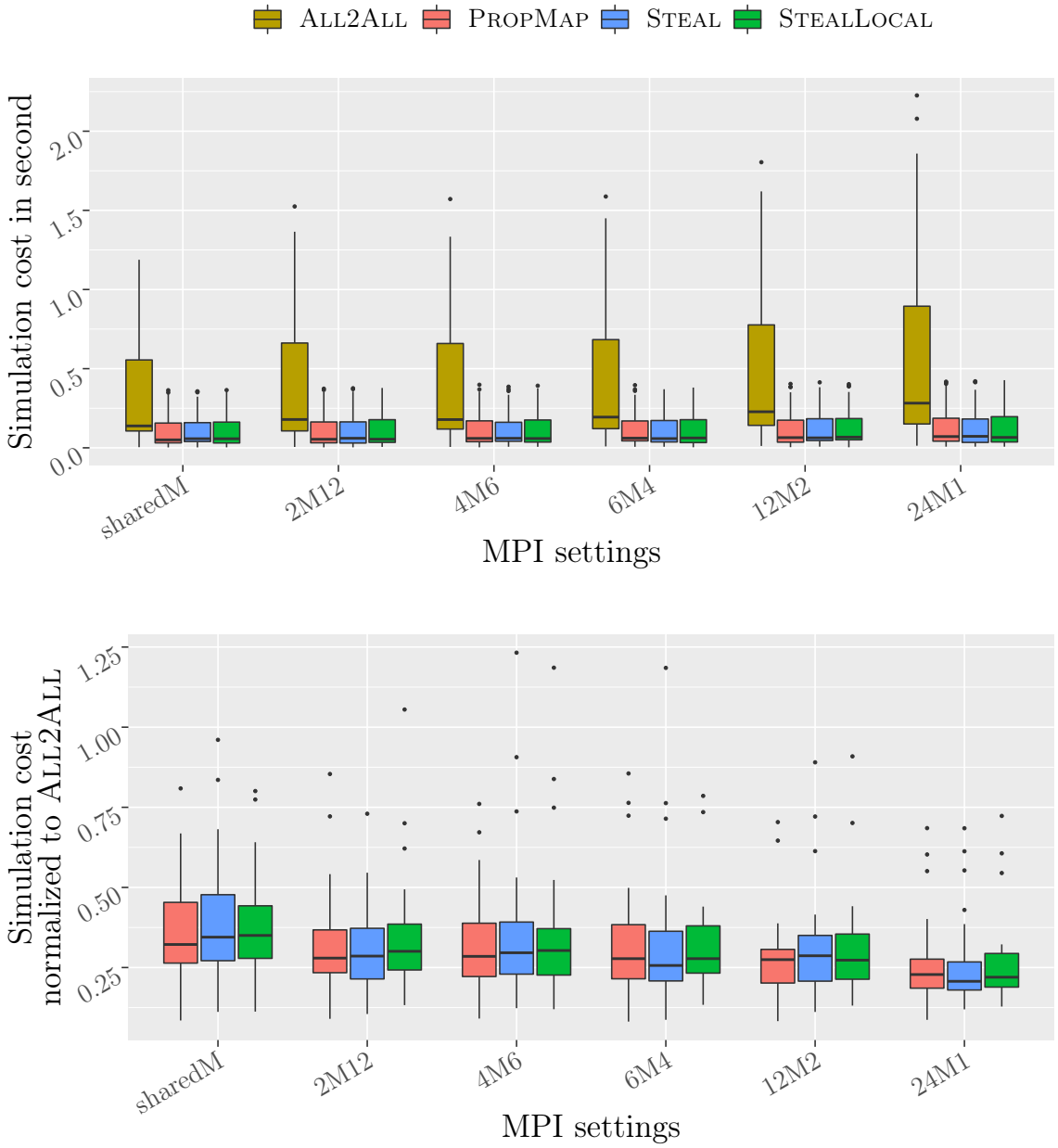


Figure 3.6 – Final simulation cost in second (*top*) and simulation cost of PropMap, Steal and StealLocal, normalized to All2All (*bottom*).

Factorization time for shared memory. Fig. 3.8 presents factorization time and its normalized version in a shared memory environment, on both `miriel` and `crunch` machines. Note that we present only the results for Steal, as StealLocal and Steal behave similarly in shared memory environment. For the sake of clarity, some large outliers are removed from the top subfigure of Fig. 3.8. On `miriel`, with a smaller number of cores and less NUMA effects, all these algorithms have almost similar factorization time, and present variations of a few tens of GFlop/s over 500GFlop/s in average. Steal slightly outperforms PropMap, and both are slower than All2All respectively by 1% and 2% in average. On `crunch`, with more cores and more NUMA effects, the difference between Steal and PropMap increases in favor of Steal. Both remain slightly behind All2All, respectively by 2% and 4%; indeed, All2All outperforms them since it has the greatest flexibility, and communications have less impact in a shared memory environment.

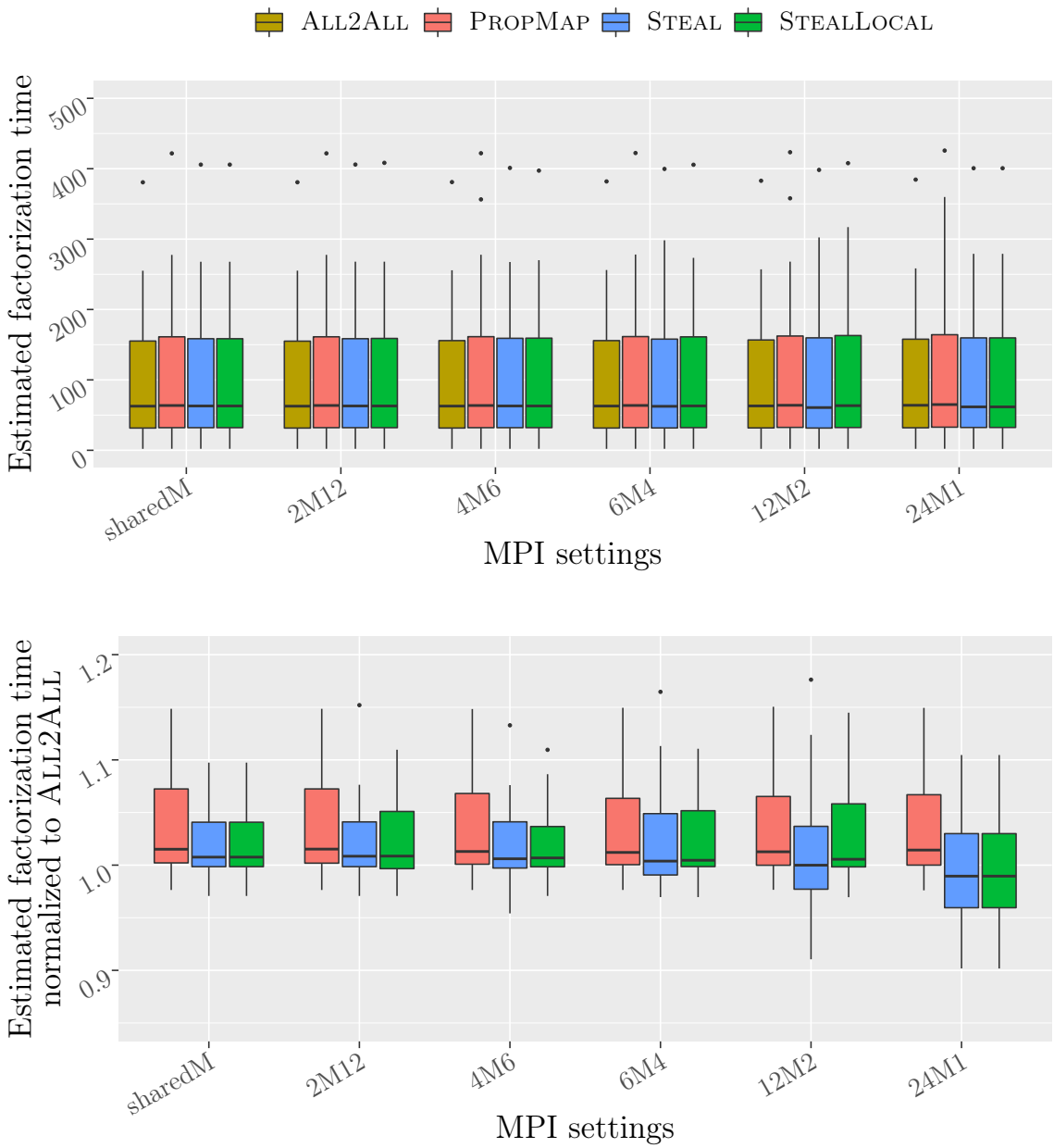


Figure 3.7 – Estimated factorization time in second (*top*), and that of PropMap, Steal, and StealLocal, normalized to All2All (*bottom*).

3.6 Chapter summary

In this chapter, we revisit the classical mapping and scheduling strategies for sparse direct solvers. The goal is to efficiently schedule the task graph corresponding to an elimination tree, so that the factorization time can be minimized. Thus, we aim at finding a trade-off between data locality (focus of the traditional PropMap strategy) and load balancing (focus of the All2All strategy). First, we improve upon PropMap by proposing a refined (and optimal) mapping strategy with an integer number of processors. Next, we design a new heuristic, Steal, together with a variant StealLocal, which predicts processor idle times in PropMap and assigns tasks to idle processors. This leads to a limited loss of locality, but improves the load balance of PropMap.

Extensive experimental and simulation results, both on shared memory and distributed memory settings, demonstrate that the Steal approach generates almost the same number of data movements than PropMap, hence the loss in locality is not significant, while it leads to better simulated factorization times, very close to that of All2All, hence improving the load balance of the schedule.

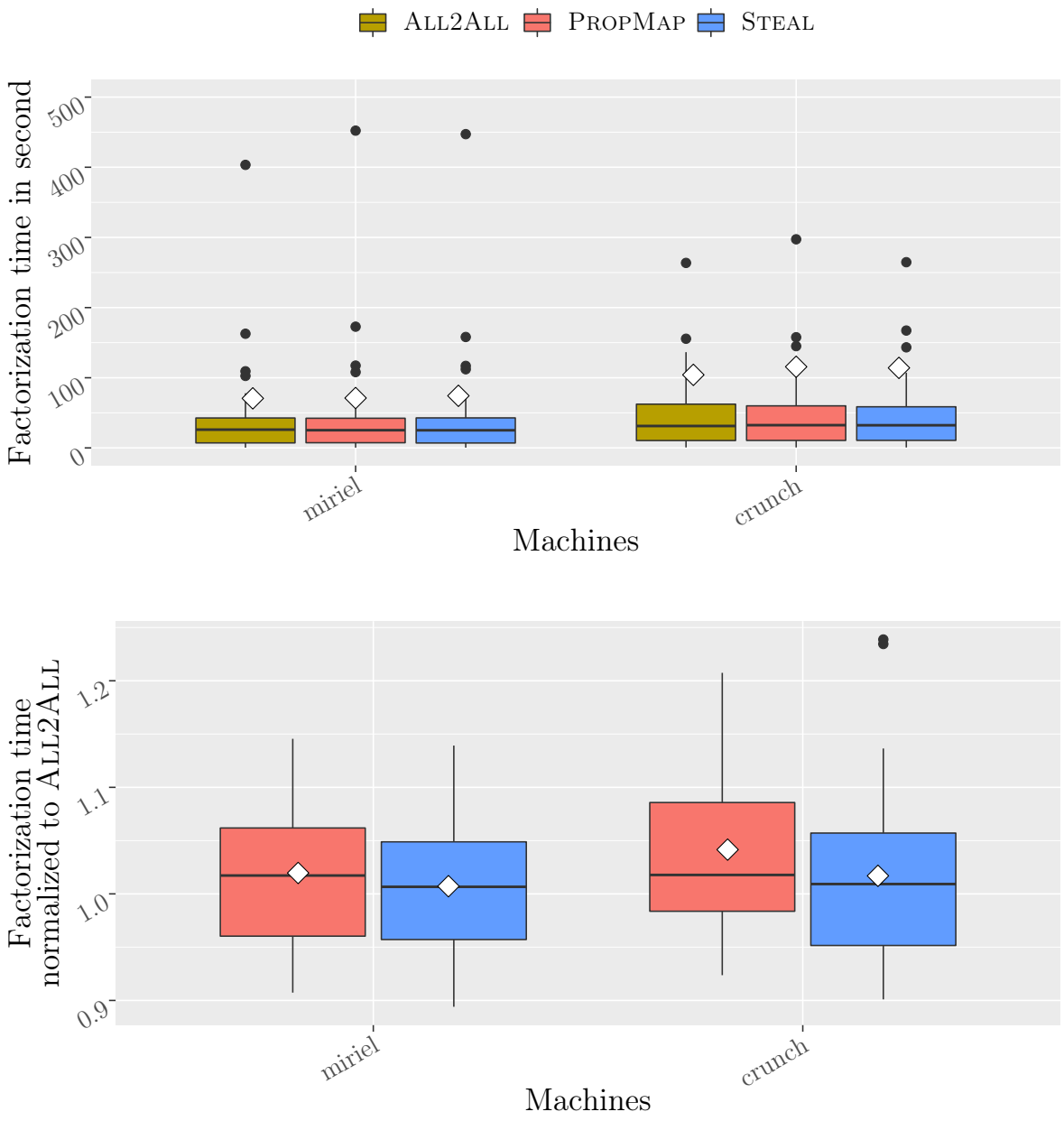


Figure 3.8 – Factorization time (*top*), and that of PropMap and Steal, normalized to All2All (*bottom*), on `miriel` and `crunch`. White diamonds represent mean values.

Chapter 4

Reliability-aware energy optimization for throughput-constrained applications on MPSoC

This chapter is a joint work with Mingsong Chen and Tongquan Wei from ECNU (Shanghai, China). Compared to applications considered in previous two chapters, streaming applications in this chapter are represented as task graphs as well, but they are more time sensitive. That is to say, the system has to run at a high speed to meet the throughput bound, otherwise, the system fails. They are often running on embedded systems, which have a tight energy budget because of their size-limited batteries. Therefore, scheduling streaming applications on these platforms brings up a multi-objective optimization problem: minimizing energy consumption under performance and reliability constraints. We proved that variants of this multi-criteria problem are NP-hard, and we proposed some efficient heuristics. This work has been published at ICPADS 2018 [C3].

4.1 Introduction

Many streaming applications in areas such as Internet of Things (IoT), augmented reality, and robotics, increasingly require high performance on embedded processing platforms. The three main criteria are i) computational performance, expressed as the throughput of the application; ii) reliability, i.e., most data sets must be successfully computed; and iii) energy efficiency. This is mainly because: i) some applications such as audio/video coding or deep learning-based inference are delay-sensitive, hence throughput should be properly guaranteed; ii) emerging safety-critical applications such as self-driving vehicles and tactile internet impose extremely stringent reliability requirements [41]; and iii) devices on which streaming applications are running are often battery-operated, hence systems should be energy-efficient.

In order to meet all these design constraints, MPSoC is becoming a new paradigm that enables effective and efficient design of streaming applications. By integrating multiple cores together with an interconnection fabric (e.g., Network-on-Chip) as communication backbone, MPSoC (e.g., *OMPA* from Texas Instruments and *NORMADIC* from STMicroelectronics) can be tailored as multiple application-specific processors with high throughputs but low energy consumption [15, 90].

As one of the most effective power management techniques, Dynamic Voltage and Frequency Scaling (DVFS) has been widely used by modern MPSoCs [25]. By properly lowering the processing voltages and frequencies of dedicatedly mapped tasks, DVFS enables streaming applications to be carried out with a reduced energy consumption, while ensuring a given throughput. However, scaling down voltages and frequencies of processors generates serious reliability problems. Various phenomena such as high energy cosmic particles and cosmic rays may cause the change of binary values held by transistors within CMOS processors by

mistake, resulting in notorious transient faults (i.e., soft errors). Along with the increasing number of transistors integrated on a chip, the susceptibility of MPSoC to transient faults will increase by several orders of magnitude [96]. In other words, the probability of incorrect computation or system crashes will become higher due to soft errors.

To mitigate the impact of soft errors, checkpointing and task replication techniques have been widely used to ensure system reliability [45, 80]. Tasks can be replicated if they do not have an internal state, this increases their reliability as it is extremely unlikely to have errors on two or more copies. Although checkpointing and task replication techniques are promising on enhancing the system reliability, frequent utilization of such fault-tolerance mechanisms is very time or resource consuming, which will in turn cost extra energy and degrade the system throughput. Clearly, the MPSoC design objectives (i.e., energy, reliability and throughput) are three contradictory requirements when we need to decide the voltage and frequency level assignments for tasks. Although there exist dozens of approaches that can effectively handle the trade-off between energy and reliability issues, few of them consider the throughput requirement in addition, see Section 4.2. Hence, given throughput and reliability constraints, how to achieve a fault-tolerant schedule that minimizes the energy consumption for a specific DVFS enabled MPSoC platform is becoming a major challenge for designers of streaming applications.

To address the above problem, this chapter proposes a novel scheduling approach that can generate energy-efficient and soft error resilient mappings for streaming applications on a given MPSoC platform. It makes following three major contributions:

1. We propose a novel model that can formally express both performance and reliability constraints for mapping applications on MPSoCs, by bounding the expected period (for performance) and the probability of exceeding the target expected period (for reliability).

2. We prove that without performance and reliability constraints, the problem is polynomially tractable, whereas adding both constraints results in an NP-complete problem.

3. We design and evaluate novel task scheduling heuristics for reliability-aware energy optimization on MPSoCs, which enforce the constraints and aim at minimizing the energy cosumption.

The remainder of this chapter is organized as follows. Related work are discussed in Section 4.2. We then formalize the application model and optimization problem in Section 4.3. Section 4.4 studies the complexity of the problem variants, and in particular proves that the complete version of the problem is NP-complete. To quickly achieve efficient mappings, Section 4.5 presents the details of our heuristic approaches. Section 4.6 conducts the evaluation of our approaches on both real and synthetic applications. Finally, Section 4.7 concludes and provides directions for future work.

4.2 Related work

To satisfy the intensive computing power demand, instead of increasing the frequency of a single core, high performance is achieved in MPSoC through dozens or even hundreds of small-sized cores, which are connected by a high-speed communication infrastructure. For instance, AsAp2 [86] consists of 164 identical programmable processors with independent clock domains. CoreVA-MPSoC [10] consists of a Network-on-Chip (NoC) interconnect that couples several processor clusters. Within each cluster, several cores are tightly coupled via a bus-based interconnect. These MPSoCs are especially designed for streaming applications, e.g., digital signal processing, multimedia processing, and autonomous navigations, in embedded and energy-limited systems [86, 10].

Throughput maximization problem has been a subject of continuing interest as the demands for MPSoC-enabled high performance computing drastically increase. Zhang et al. [95] optimized the throughput in disruption tolerant networks via distributed workload dissemination, and designed a centralized polynomial-time dissemination algorithm based on the shortest delay tree. Li et al. [59] specifically considered stochastic characteristics of task execution time to trade off schedule length (i.e., throughput) for energy consumption. A novel Monte Carlo based task scheduling is developed to generate a static schedule that can maximize the expected throughput without incurring a prohibitively high time overhead [97]. Albers et al. [3] introduced an online algorithm to further maximize throughput with parallel schedules. Qi et at. [83] proposed two Integer Linear Programming formulations with an explicit emphasis on avoiding potential communication contention in scheduling a DAG. In their work, duplication of tasks allows to ease the communication overhead and to minimize the schedule length. Wayne et al. [53] presented a novel simulated-annealing based partitioning algorithm to map streaming applications onto a hierarchical MPSoC. However, reliability issues are not considered in these works.

Reliability can be achieved by reserving some CPU time for re-executing faulty tasks due to soft errors [96]. Marwedel et al. present a representative set of techniques in [67] that map embedded applications onto multicore architectures. These techniques focus on optimizing performance, temperature distribution, reliability and fault tolerance for various models. Dongarra et al. [35] studied the problem of scheduling task graphs on a set of heterogeneous resources to maximize reliability and throughput, and proposed a throughput/reliability tradeoff strategy. Wang et al. [89] developed a look-ahead genetic algorithm to optimize both the system reliability and throughput for distributed workflow application. Wang et al. [88] proposed replication-based scheduling for maximizing system reliability. The proposed algorithm incorporates task communication into system reliability and maximizes communication reliability by searching all optimal reliability communication paths for current tasks. These works explore the reliability of heterogeneous multicore processors from various aspects, and present efficient reliability improvement schemes, however, these works do not investigate the energy consumed by MPSoCs, which interplays with system reliability.

Extensive research effort has been devoted to reduce energy consumption of DVFS-enabled heterogeneous multi-core platforms considering system reliability. Zhang et al. [94] proposed a novel genetic algorithm based approach to improve system reliability in addition to energy savings for scheduling workflows in heterogeneous multicore systems. In [80], Spasic et al. presented a novel polynomial-time energy minimization mapping approach for synchronous dataflow graphs. They used task replication to achieve load-balancing on homogeneous processors, which enables processors to run at a lower frequency and consume less energy. Vilches et al. [87] considered mapping streaming application onto a heterogeneous embedded system that consists of multi-core CPU and on-chip GPU. Assigning the same task to a CPU or a GPU leads to different execution speeds and synchronization overheads. A two steps framework is proposed to adaptively find the optimal throughput or energy, or a trade-off of both: a training phase followed by a running phase. The aim of the training phase is to collect energy cost and execution speed of the basic mapping and to decide the optimal configuration. In [69], Onnebrink et al. mapped each task to a processing element of a heterogeneous MPSoC and selected its execution frequency so that the energy cost is minimized under given makespan and fixed mapping constraints. In [27], Das et al. proposed a genetic algorithm to improve the reliability of DVFS-based MPSoC platforms while fulfilling the energy budget and the performance constraint. However, their task mapping approach tries to minimize core aging together with the susceptibility to transient errors. Haque et al. considered in [45] the problem of achieving a given reliability target for a set of periodic real-time tasks running on a multicore system with minimum energy consumption. The proposed framework explicitly takes into account the coverage factor of the fault detection techniques and the negative impact of DVFS on the rate of transient

faults leading to soft errors.

Although above works explore various techniques to save energy, to the best of our knowledge, none of the above works considers system throughput in addition to reliability and energy. Our approach is thus the first attempt to model both reliability, performance and energy for workflow scheduling in MPSoC.

4.3 Models and optimization problems

We consider the problem of scheduling a pipelined workflow onto a homogeneous multi-core platform that is subject to failures. The goal is to minimize the expected energy consumption for executing a single dataset, given some constraints on the expected and worst-case throughput of the workflow. In the following subsections, we detail how to model applications, platforms, failures, energy cost, period (which is the inverse of the throughput), and how to formally define the optimization problem.

4.3.1 Streaming applications – linear chain

We focus on linear chain workflow applications, where task dependencies form a linear chain: each task requests an input from the previous task, and delivers an output to the next task. There are n tasks $T_1, \ldots, T_n$. Furthermore, the application is pipelined, i.e., datasets continuously enter through the first task, and several datasets can be processed concurrently by different tasks. Such applications are ubiquitous in processing of streaming datasets in the context of embedded systems [49].

We assume that the initial data resides in memory, and the final data stays in memory. Task T_i is characterized by a workload w_i, and the size of its output file to the next task $o_{i,i+1}$, as illustrated in Fig. 4.1 (except for the last task). In the example, we have $w_1 = 2$, $w_2 = 5$, $w_3 = 4$, and $o_{1,2} = 3$, $o_{2,3} = 1$. Once the first dataset reaches task T_3, while it is processed by T_3, dataset 2 is transferred between T_2 and T_3, dataset 3 is processed by T_2, dataset 4 is transferred between T_1 and T_2, and dataset 5 is processed by T_1. At the next period, all dataset numbers are incremented by one.

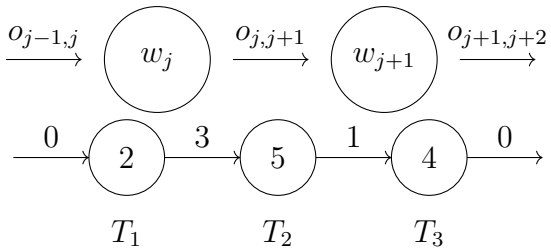


Figure 4.1 – Linear chain workflow application.

4.3.2 Platforms

The target platforms are embedded systems composed of p homogeneous computing cores. Each core can run at a different speed with a corresponding error rate and an energy consumption. If task T_i is executed on a core operating at speed $s(i)$ and if it is not subject to a failure, it takes a time $\frac{w_i}{s(i)}$ to execute a single dataset.

We focus on the most widely used speed model, the *discrete* model, where cores have a discrete number of predefined speeds, which correspond to different voltages at which the core can be operating. Switching is not allowed during the execution of a given task, but two different executions of a task can be executed at different speeds. The set of speeds is $\{s_{\min} = s_1, s_2, \ldots, s_k = s_{\max}\}$. The *continuous* model is used mainly for theoretical studies,

and let the speed take any value between the minimum speed $s_{\min}$ and the maximum speed $s_{\max}$.

All cores are fully interconnected by a NoC. The bandwidth β is the same between any two cores, hence it takes $\frac{o_{i,i+1}}{\beta}$ for task T_i to communicate a dataset to task T_{i+1}. The NoC enables cores to communicate simultaneously with others while they are computing, i.e., communications and computations can be overlapped. Therefore, while task T_i is processing dataset k, it is receiving the input for dataset $k-1$ from the previous task, and sending the output for dataset $k+1$ to the next task. Hence, these operations overlap, and take respectively a time $\frac{o_{i-1,i}}{\beta}$ and $\frac{o_{i,i+1}}{\beta}$.

We follow the model of [48, 17], where cores are equipped with a router, and on which there are registers. We can use the registers to store intermediate datasets, hence having *buffers* between cores. If datasets are already stored in the input buffer of a core, and if there is empty space in the output buffer, then the core can process a dataset without having to wait for the previous or next core.

4.3.3 Failure model and duplication

Embedded system platforms are subject to failures, and in particular transient errors caused by radiation. When subject to such errors, the system can return to a safe state and repeat the computation afterwards. According to the work of [98], radiation-induced transient failures follow a Poisson distribution. The fault rate is given by:

$$\lambda(s) = \lambda_0 e^{d\frac{s_{\max}-s}{s_{\max}-s_{\min}}},$$

where $s \in [s_{\min}, s_{\max}]$ denotes the running speed, d is a constant that indicates the sensitivity to dynamic voltage and frequency scaling, and λ_0 is the average failure rate at speed $s_{\max}$. λ_0 is usually very small, of the order of 10^{-5} per hour [7]. Therefore, we can assume that there are no failures when running at speed $s_{\max}$. We can see that a very small decrease of speed leads to an exponential increase of failure rate.

The failure probability of executing task T_i (without duplication) on a processor running at speed s_k is therefore $f_i(s_k) = \lambda(s_k)\frac{w_i}{s_k}$. If an error strikes, we resume the execution by reading the dataset again from local memory (i.e., the input has been copied before executing the task, we re-execute the task on the copy), and this re-execution is done at maximum speed so that no further error will strike the same dataset on this task. We assume that the time to prepare re-execution is negligible. Still, this slows down the whole workflow since other tasks may need to wait.

We propose to duplicate some tasks to mitigate the effect of failures and have a reliable execution. This means that two identical copies of a same task are executed on two distinct cores, both core running at the same speed. In this case, if a failure occurs in only one copy, we can keep going with the successful copy. However, it may increase the energy cost and communication cost. Similarly to one execution at the maximum speed, we assume that an error on a duplicated task is very unlikely (i.e., at least one copy will be successful), and hence $f_i(s_k) = 0$ if T_i is duplicated.

Let $m_i = 1$ if task T_i is duplicated, and $m_i = 0$ otherwise. Let s_k be the speed at which T_i is processed. The failure probability for T_i is therefore $f_i(s_k) = (1-m_i)\lambda(s_k)\frac{w_i}{s_k}$, i.e., it is zero if the task is duplicated, and $\lambda(s_k)\frac{w_i}{s_k}$ otherwise (the instantaneous error rate at speed s_k times the time to execute task T_i).

If we do not account for communications, the expected execution time of task T_i running at speed s_k is:

$$t_i = \frac{w_i}{s_k} + f_i(s_k)\frac{w_i}{s_{\max}}.$$

Indeed, with duplication, at least one execution will be successful, while with a single execution, if there is a failure, we re-execute the task at maximum speed and there are no further failures.

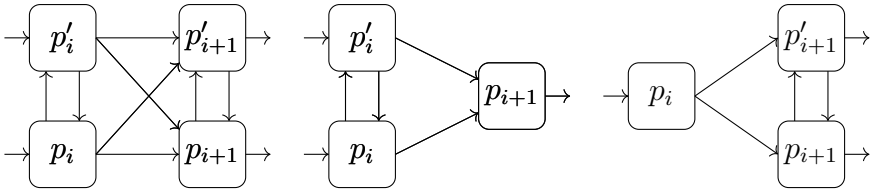


Figure 4.2 – Communications with task duplication.

If a task is duplicated, this implies that further communications may be done, but they will occur in parallel. If T_i is duplicated, both processors p_i and p'_i on which T_i is executed are synchronized, and only one of them obtaining a correct result will do the output communication (to one or two processors, depending on whether T_{i+1} is duplicated or not), see Fig. 4.2 for different possible configurations. The synchronization cost is assumed to be negligible.

4.3.4 Energy

We follow a classical energy model, see for instance [11], where the dissipated power for running at speed s_k is s_k^3, and hence the energy consumed for a single execution of task T_i running at speed s_k is $\frac{w_i}{s_k} \times s_k^3 = w_i s_k^2$. We further account for possible failures and duplication, hence obtaining the expected energy consumption for T_i running at speed s_k for one dataset:

$$E_i(s_k) = (m_i + 1)w_i s_k^2 + f_i(s_k)w_i s_{\max}^2.$$

Indeed, if task T_i is duplicated ($m_i = 1$), we always pay for two executions ($2w_i s_k^2$) but there is no energy consumed following a failure, while without duplication ($m_i = 0$), we account for the energy consumed by the re-execution in case of a failure.

We assume that the energy consumed by communications and buffers is negligible compared to the energy consumed by computations, see [48]. Therefore, the expected energy consumption of the whole workflow to compute a single dataset is the sum of the expected energy consumption of all tasks.

4.3.5 Period definition and constraints

In this chapter, each task is mapped onto a different processor, or a pair of processors when duplicated, and different tasks are processing different datasets. In steady-state mode, the throughput is either constrained by the task with the longest execution time, or by the longest communication time, which is slowing down the whole workflow. The time required between the execution of two consecutive datasets corresponds to this bottleneck time and is called the period. It is the inverse of the throughput.

In this work, we are given a target period P_{t}, hence the target throughput is $\frac{1}{P_{\mathrm{t}}}$. This corresponds for instance to the rate at which datasets are produced. We consider two different constraints: i) ensure that the expected period is not exceeding P_{t}, hence the target becomes a bound, and/or ii) ensure that the probability of exceeding the target P_{t} for a given dataset is not greater than $proba_{\mathrm{t}}$. This second constraint corresponds to real-time systems, where a dataset is lost if its execution exceeds the target period P_{t}, and the probability $proba_{\mathrm{t}}$ ($0 \leq proba_{\mathrm{t}} \leq 1$) expresses how many losses are tolerated. If $proba_{\mathrm{t}} = 1$, there is no constraint, while $proba_{\mathrm{t}} = 0$ means that no losses are tolerated.

Recall that the objective is to minimize the expected energy consumption per dataset of the whole workflow. Some tasks may be duplicated, and each task may run at a different speed. The communication between two consecutive tasks T_i and T_{i+1} takes a constant time $\frac{o_{i,i+1}}{\beta}$, and it must fit within the target period. Therefore, we assume that for all $1 \leq i < n$, $\frac{o_{i,i+1}}{\beta} \leq P_{\mathrm{t}}$.

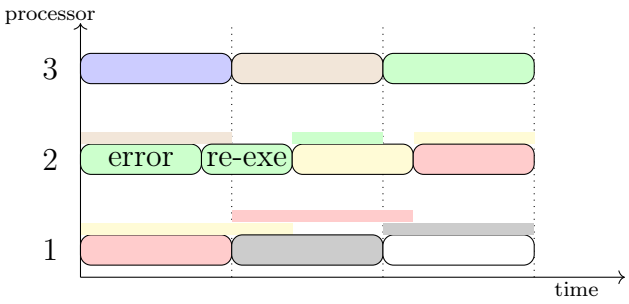


Figure 4.3 – An instance illustrates fault tolerance with buffers. Rectangles with rounded corners represent tasks running on different processors and other rectangles between them represent buffers. Only buffers in-use are depicted. Datasets are labeled by colors. error and re-exe represent respectively an error happened and the re-execution afterwards. The vertical dashed lines indicate the start or end of a period. Communication is not depicted here.

We assume in this section that the set of duplicated tasks is know: we set m_i to 0 or 1 for each task T_i. Furthermore, let $s(i)$ be the speed at which task T_i is executed, for $1 \leq i \leq n$.

In Section 4.3.5, we first consider the case without failures and express the period in this case. Then, we express the expected period when the platform is subject to failures. Note that we assume that there is a sufficient number of buffers between cores, so that a failure does not necessarily impact the period, given that the cores have access to datasets stored in buffers, and can use empty buffers to store output datasets. Finally, we explain how to compute the probability that a dataset exceeds the target period P_{t}.

Period without failures

In the case without failures, the period is determined by the bottleneck task computation or communication: $P_{\mathrm{nf}} = \max_{1 \leq i \leq n} \left\{ \frac{w_i}{s(i)}, \frac{o_{i,i+1}}{\beta} \right\}$.

We denote by L the set of tasks whose execution time is equal to P_{nf}, i.e., $L = \left\{ T_i | \frac{w_i}{s(i)} = P_{\mathrm{nf}} \right\}$. If the bottleneck time P_{nf} is achieved by a communication, this set may be empty.

Expected period

We consider that each processor is equipped with three or more buffers, two of them holding an input (resp. output) dataset being received (resp. sent), and the other buffers are used for storing intermediate datasets: a buffer is filled when the task is completed, but the following processor is not yet ready to receive the next dataset (i.e., the output buffer is still in use). We consider the period in steady-state, after the initialization has been done, i.e., all processors are currently working on some datasets.

The set of tasks L is empty if the computation time for all tasks is strictly smaller than the period P_{nf}. When subject to a failure, tasks not in L can use data stored in buffers and process datasets at a faster pace than the period, until they have caught up with the time lost due to the failure.

Figure 4.3 provides an example, where the task T_2 running on processor 2, which is not in L, failed on the green dataset. At the beginning of the next period, the task T_3 running on processor 3 can read the input from the buffer and it continues at a period smaller than P_{nf} until it has caught up, and the overall period remains P_{nf}.

However, errors in tasks of L are impacting the period, and therefore, if such a task is subject to a failure, the re-execution time is added to the period. The expected period can

therefore be expressed as follows:

$$P_{\text{exp}} = P_{\text{nf}} + \sum_{i \in L} f_i(s(i)) \frac{w_i}{s_{\max}}. \tag{4.1}$$

Indeed, the period P_{nf} is achieved when there is either no failure, or a failure in a task not in L. In case of a failure while executing a task T_i in L, the period is $P_{\text{nf}} + \frac{w_i}{s_{\max}}$, and this happens with a probability $f_i(s(i))$. As discussed before, we assume that there is no failure during re-execution, and that the probability of having two failures while executing a single dataset is negligible.

Note that this formula also holds when some tasks are duplicated. If task $i \in L$ is duplicated ($m_i = 1$), it will never fail and hence its period will be P_{nf}. In this case, $f_i(s(i)) = 0$ by definition, hence the formula remains correct.

Bounding the probability of exceeding the period bound

For the second constraint, we focus on the actual period of each dataset, rather than the expected period, and we estimate the probability at which the period of a dataset exceeds P_{t}. We consider that $P_{\text{nf}} \leq P_{\text{t}}$, otherwise the bound can never be reached, and the probability is always one.

The actual period, denoted by P_{act}, is a random variable that ranges from P_{nf} to $\max_{1 \leq i \leq n} \left(\frac{w_i}{s(i)} + \frac{w_i}{s_{\max}} \right)$. We define the set of tasks that may exceed the target period P_{t} in case of failures:

$$S_{\text{excess}} = \left\{ T_i \;\middle|\; \frac{w_i}{s(i)} + \frac{w_i}{s_{\max}} > P_{\text{t}} \right\}.$$

Therefore, if a failure strikes a task in S_{excess} on a given dataset, the target period P_{t} may not be met for this dataset. An error happens on task i with probability $f_i(s(i))$. Since failures are independent, the period of a dataset will not exceed the bound if and only if no task in the set S_{excess} has a failure, i.e., this happens with a probability $\prod_{T_i \in S_{\text{excess}}} (1 - f_i(s(i)))$.

Hence, the probability of exceeding the bound is given by:

$$P(P_{\text{act}} > P_{\text{t}}) = 1 - \prod_{T_i \in S_{\text{excess}}} (1 - f_i(s(i))) \approx \sum_{T_i \in S_{\text{excess}}} f_i(s(i)), \tag{4.2}$$

considering that the failure probabilities are small, and that $f_i(s(i)) \times f_j(s(j)) = 0$ for any $1 \leq i, j \leq n$. This approximation is in line with the assumption that we do not consider two consecutive failures in a same task.

Finally, the second constraint that we consider, after the one on the expected period described above, is to bound the probability of exceeding the target period P_{t} by the target probability $proba_{\text{t}}$:

$$P(P_{\text{act}} > P_{\text{t}}) \leq proba_{\text{t}}.$$

4.3.6 Optimization problem

The objective is to minimize the expected energy consumption per dataset of the whole workflow, and we consider two constraints. The goal is to decide which tasks to duplicate, and at which speed to operate each task. More formally, the problem is defined as follows:

(MinEnergy). *Given a linear chain composed of n tasks, a computing platform with p homogeneous cores that can be operated with a speed within set S, a failure rate function f, and a target period P_t, the goal is to decide, for each task T_i, whether to duplicate it or not (set $m_i = 0$ or $m_i = 1$), and at which speed to operate it (choose $s(i) \in S$), so that the total expected energy consumption is minimized, under the following constraints:*
i) The expected period P_{exp} should not exceed P_t;

ii) The probability of exceeding the target period P_t should not exceed the target probability $proba_t$.

Note that if there is a task k such that $P_\mathrm{t} < \frac{w_k}{s_\mathrm{max}}$ or $P_\mathrm{t} < \frac{o_{k,k+1}}{\beta}$, then there is no solution since the target period can never be met.

If P_t is large enough, the problem will not be constrained since in all solutions, the expected period will always be under the target period. This is the case for $P_\mathrm{t} \geq \max(\frac{w_i}{s_\mathrm{min}} + \frac{w_i}{s_\mathrm{max}})$ and $P_\mathrm{t} \geq \max(\frac{o_{k,k+1}}{\beta})$. In this case, each task running at the slowest possible speed, and being re-executed after a failure, will not exceed P_t. This problem without constraints is denoted as MinEnergy-NoC.

We also consider the particular cases where only one or the other constraint matters. MinEnergy-PerC is the problem where we do only consider the first constraint on the expected period (i.e., set $proba_\mathrm{t} = 1$), while MinEnergy-ProbaC is the problem where we do only focus on the probability of exceeding the target period, i.e., we do not consider P_exp.

4.4 Complexity analysis

4.4.1 Without errors

When the workflow is free of errors, a task T_i running at speed s_i takes exactly a time $\frac{w_i}{s_i}$, and consumes an energy of $w_i s_i^2$. Hence, to minimize the energy consumption, one must use the smallest possible speed such that the target period is not exceeded, hence $s_i = \max\left\{\frac{w_i}{P_\mathrm{t}}, s_\mathrm{min}\right\}$ in the *continuous* case. Since we consider *discrete* speeds, the optimal speed for task T_i is therefore the smallest speed larger than or equal to $\frac{w_i}{P_\mathrm{t}}$ within the set of possible speeds. This is true for all tasks, hence the problem can be solved in polynomial time.

4.4.2 Without constraints

We consider the MinEnergy-NoC problem, and propose the BestEnergy algorithm to optimally solve this problem. The idea is to use the speed that minimizes the energy consumption for each task, since we do not have any constraint about exceeding the target period. For each task, either we execute it at this optimal speed, or it may be even better (in terms of energy consumption) to duplicate it and run it at the smallest possible speed s_min.

Theorem 2. MinEnergy-NoC *can be solved in polynomial time, using the* BestEnergy *algorithm, both for the* discrete *and for the* continuous *energy model.*

Proof. Given a task of weight w executed at speed s without duplication, the energy consumption is $E(s) = w \times s^2 + \lambda_0 w^2 s_\mathrm{max}^2 \frac{e^{d\frac{s_\mathrm{max}-s}{s_\mathrm{max}-s_\mathrm{min}}}}{s}$. The (continuous) speed that minimizes this energy consumption can be obtained by deriving $E(s)$:

$$E'(s) = 2ws - \left(\frac{\lambda_0 w^2 s_\mathrm{max}^2}{s^2} + \frac{\lambda_0 d w^2 s_\mathrm{max}^2}{s(s_\mathrm{max} - s_\mathrm{min})}\right) e^{d\frac{s_\mathrm{max}-s}{s_\mathrm{max}-s_\mathrm{min}}}.$$

$E'(s)$ is a monotonically increasing function, and we let s^* be the speed such that $E'(s^*) = 0$, hence $E(s^*)$ is minimum.

If the task is not duplicated, for MinEnergy-NoC-Cont, the optimal speed is $s_\mathrm{opt} = \max\{s^*, s_\mathrm{min}\}$. In the *discrete* case MinEnergy-NoC-Disc, s_opt is simply the speed that minimizes the energy consumption, hence $s_\mathrm{opt} = \mathrm{argmin}_{s\in\{s_\mathrm{min},\ldots,s_\mathrm{max}\}}\{E(s)\}$.

Now, if the task is duplicated, we assume that it will not be subject to error, hence the energy consumption at speed s is $2ws^2$. Therefore, it is minimum when the task is executed at the minimum speed, and the corresponding energy consumption is $2ws_\mathrm{min}^2$ (both in the *discrete* and continuous case).

BestEnergy is a greedy algorithm that sets the speed of each task at s_{opt} (not using duplication), and then greedily assigns remaining processors to tasks that would gain most from being duplicated (if any), see Algorithm 10. It is easy to see that it is optimal, since any other solutions could only have a greater energy consumption. □

Algorithm 10 – BestEnergy (n, p)

```
for i = 1 to n do
    Compute s_opt(i) for task T_i, the speed that minimizes energy consumption if T_i is not duplicated;
    s_i ← s_opt(i), m_i ← 0;
    g_i ← E(s_i) − 2 w_i s_min^2 (Possible gain in energy if T_i is duplicated);
end for
Sort tasks by non-increasing g_i, T_j is the task with max g_i;
p_av ← p − n (Number of available processors);
while g_j > 0 and p_av > 0 do
    m_j ← 1, s_j ← s_min, p_av ← p_av − 1;
    j ← the index of next task in the sorted list;
end while
return < s_i, m_i >;
```

4.4.3 With the probability constraint

We now prove that the decision version of MinEnergy-ProbaC is NP-complete. In the decision version of MinE-Dec, the goal is to find an assignment set of speeds such that the probability of exceeding the target period P_{t} does not exceed $proba_{\text{t}}$, and such that the energy consumption does not exceed a given energy threshold E_{t}. The proof is based on a reduction from the Partition problem, known to be NP-complete [43]. The idea is to have only two possible speeds, and one must decide at which speed to operate each task. We set as many processors as tasks, so that no duplication can be done.

Theorem 3. MinE-Dec *is NP-complete, even when duplicating tasks is not possible.*

Proof. We first check that MinE-Dec is in NP: given a speed for each task, it is easy to verify in polynomial time whether the bounds on the failure probability and on the energy consumption are satisfied.

The proof of completeness is based on a reduction from the Partition problem, known to be NP-complete [43]. We consider an instance $\mathcal{I}_{1-\text{Par}}$ of 2-partition: given n positive integers $a_1, \ldots, a_n$, does there exist a partition of $\{1, \ldots, n\}$ into two subsets I_1 and I_2 ($I_1 \cup I_2 = \{1, \ldots, n\}$ and $I_1 \cap I_2 = \emptyset$) such that $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i = S/2$, where $S = \sum_{i=1}^{n} a_i$?

We let $\Delta = \max a_j / \min a_j$. We build an instance $\mathcal{I}_{2-\text{MinE}}$ of MinE-Dec as follows:

- The workflow is made of n tasks, of size $w_1 = a_1, \ldots, w_n = a_n$, to be processed on $p = n$ cores (no duplication is possible).
- There are only two possible speeds, $s_{\min} = s_1 = 1$, and $s_{\max} = s_2 = 2\Delta$.
- The failure rate function for these speeds is given by $f(s_1) = 1/S$ and $f(s_2) = 0$.
- We set the target period to $P_{\text{t}} = \min_i w_i$, the bound on the probability of exceeding P_{t} to $proba_{\text{t}} = 1/2$, and the bound on the energy to $E_{\text{t}} = 2\Delta^2(S+1) + S/2$.

We first assume that there is a solution (I_1, I_2) to instance $\mathcal{I}_{1-\text{Par}}$. For the MinE-Dec problem, we set all tasks T_i with $i \in I_1$ to speed $s_1 = 1$, and all tasks T_i with $i \in I_2$ to speed $s_2 = 2\Delta$. Given the target period, we check that $S_{\text{excess}} = I_1$:

- For any task in I_1, we have $w_i/s_1 + w_i/s_{\max} > w_i \geq \min w_j = P_{\mathrm{t}}$.
- For any task in I_2, we have $w_i/s_2 + w_i/s_{\max} = 2w_i/(2\Delta) = w_i \min w_j / \max w_i \leq \min w_j = P_{\mathrm{t}}$.

Thus, the probability of exceeding P_{t} is given by

$$\sum_{i \in I_1} f(s_1) w_i/s_1 = 1/S \sum_{i \in I_1} w_i = 1/S \times S/2 = 1/2 = \mathit{proba}_{\mathrm{t}},$$

which satisfies the constraint on the probability. Then, we compute the energy of the obtained solution:

$$\begin{aligned} E &= \sum_{i \in I_1} (w_i s_1^2 + f(s_1) w_i s_{\max}^2) + \sum_{i \in I_2} w_i s_2^2 \\ &= S/2(1 + 4\Delta^2/S) + S/2 \cdot 4\Delta^2 = E_{\mathrm{t}}, \end{aligned}$$

which satisfies the bound on the energy. Hence, we have found a solution to $\mathcal{I}_{2-\mathrm{MinE}}$.

We then assume that $\mathcal{I}_{2-\mathrm{MinE}}$ has a solution. We denote by I_1 the set of tasks running at speed s_1, and by I_2 the others, running at speed s_2. As outlined below, only tasks in I_1 contribute to the probability of exceeding P_{t}, and its bound writes:

$$\sum_{i \in I_1} f(s_1) w_i/s_1 \leq 1/2$$

With $s_1 = 1$ and $f(s_1) = 1/S$, this gives $\sum_{i \in I_1} w_i \leq S/2$. The bound on the energy writes:

$$\begin{aligned} \sum_{i \in I_1} (w_i s_1^2 + f(s_1) w_i s_{\max}^2) + \sum_{i \in I_2} w_i s_2^2 &\leq 2\Delta^2(S+1) + S/2 \\ \sum_i w_i s_1^2 + \sum_{i \in I_1} w_i/S s_{\max}^2 + \sum_{i \in I_2} w_i (s_2^2 - s_1^2) &\leq 2\Delta^2(S+1) + S/2 \\ S + \sum_{i \in I_1} w_i/S s_{\max}^2 + \sum_{i \in I_2} w_i (4\Delta^2 - 1) &\leq 2\Delta^2(S+1) + S/2 \end{aligned}$$

$$\begin{aligned} \sum_{i \in I_2} w_i (4\Delta^2 - 1) &\leq 2\Delta^2(S+1) - S/2 - \sum_{i \in I_1} w_i/S s_{\max}^2 \\ \sum_{i \in I_2} w_i (4\Delta^2 - 1) &\leq 2\Delta^2(S+1) - S/2 \\ \sum_{i \in I_2} w_i (4\Delta^2 - 1) &\leq S/2(4\Delta^2 - 1) + 2\Delta^2 \\ \sum_{i \in I_2} w_i &\leq S/2 + \frac{2\Delta^2}{4\Delta^2 - 1} \end{aligned}$$

Since $2\Delta^2/(4\Delta^2 - 1) < 1$ as soon as $\Delta \geq 1$ and all w_i's are integers, this gives $\sum_{i \in I_2} w_i \leq S/2$. Together with $\sum_{i \in I_1} \leq S/2$, this proves that I_1, I_2 is a solution to $\mathcal{I}_{1-\mathrm{Par}}$, which concludes the proof. □

4.5 Heuristics

We start with basic heuristics that will be used as baseline. Then we design heuristics aiming at bounding the expected period, and finally heuristics for bounding the probability of exceeding the target period. Except baseline heuristics, all others are designed in two flavours, one for the more realistic case of *discrete* speed, and the other for the case of *continuous* speed.

4.5.1 Baseline heuristics

We first outline the baseline heuristics that will serve as a comparison point, but may not satisfy the constraints. First, the BestEnergy algorithm described in Section 4.4.2 is providing a lower bound on the energy consumption, but since it means that many tasks are running at the minimum speed, we expect the period to be large, and it may well exceed the bound.

Another simple solution consists in having each task executed at the maximum speed $s_{\max}$. We refer to this heuristic as MaxSpeed (see Algorithm 11).

Algorithm 11 – MaxSpeed (n, p)

1: **for** $i = 1$ to n **do**
2: $\quad s_i \leftarrow s_{\max}$, $m_i \leftarrow 0$;
3: **end for**
4: **return** $< s_i, m_i >$;

The third baseline heuristic, DuplicateAll (see Algorithm 12), duplicates all tasks, assuming that there are twice more processors than tasks ($p \geq 2n$), and the corresponding speeds for each tasks used in this case are the ones derived in Section 4.4.1. Indeed, there will not be any errors in this case, and we aim at respecting the target period P_{t}.

Note that both MaxSpeed and DuplicateAll will always satisfy the bounds, since there will be no errors, and hence the expected period is equal to the period without failure. However, both heuristics may lead to a large waste of energy. They provide an upper bound on the energy consumption when using a naive approach.

4.5.2 Bounding the expected period

In this section, we focus on the constraint on the expected period, hence targeting the MinEnergy-PerC problem.

Heuristic Threshold

The Threshold heuristic aims at reaching the target expected period P_{t} (see Algorithm 13) in *discrete* speed option. The first step consists in setting all task speeds to the smallest speed such that $\frac{w_i}{s_i} \leq P_{\mathrm{t}}$. If P_{exp} is still larger than P_{t}, then one of the tasks with largest duration ($\frac{w_i}{s_i}$) is duplicated: this allows P_{nf} to be smaller than P_{t} and constant from this moment on. Note that in the special case of a communication time reaching P_{t}, there is no need to duplicate a task to have $P_{\mathrm{nf}} = P_{\mathrm{t}}$.

From Equation (4.1), $P_{\mathrm{exp}} = P_{\mathrm{nf}} + \sum_{j \in L} f_j(s(j)) \frac{w_j}{s_{\max}}$. We made sure that P_{nf} is smaller than or equal to P_{t}. In order to make $P_{\mathrm{exp}} \leq P_{\mathrm{t}}$, each task T_i of L has to be either run at a higher speed (which removes it from L), or duplicated (which sets $f_{(s(i))}$ to 0). We greedily duplicate tasks for which duplication costs less energy, until there remains no

Algorithm 12 – DuplicateAll (n, p)

1: **if** $p \geq 2n$ **then**
2: $\quad$ **for** $i = 1$ to n **do**
3: $\quad\quad$ choose s_i as the smallest possible speed so that $\frac{w_i}{s_i} \leq P_{\mathrm{t}}$;
4: $\quad\quad$ $m_i \leftarrow 1$;
5: $\quad$ **end for**
6: $\quad$ **return** $< s_i, m_i >$;
7: **else**
8: $\quad$ **return** *failure*;
9: **end if**

more processors. Then, we speed up other tasks. Finally, we use the same technique as in BestEnergy to attempt to reduce again the energy of non-duplicated tasks: if the minimum speed s for energy consumption is larger than the actual speed s_i of a task T_i, its speed is increased to s.

Algorithm 13 – Threshold (n, p)

1: **for all** tasks T_i **do**
2: $s_i \leftarrow$ the smallest speed such that $\frac{w_i}{s_i} \leq P_{\text{t}}$, $m_i \leftarrow 0$;
3: **end for**
4: $p_{av} \leftarrow p - n$ (number of available processors);
5: **if** $P_{\text{t}} > \max(\frac{o_k}{\beta})$ and $p_{av} \geq 1$ **then**
6: Select a task T_k with largest duration (break tie by selecting one with smallest w_k), set $m_k \leftarrow 1$ and $p_{av} \leftarrow p_{av} - 1$;
7: **end if**
8: **if** $P_{\text{exp}} > P_{\text{t}}$ **then**
9: $Q \leftarrow \{$tasks of L with $m_i = 0\}$;
10: **for all** tasks T_j in Q **do**
11: $s \leftarrow$ the smallest speed that is larger than s_j;
12: $g_j \leftarrow w_j * (s^2 + f_j(s)s_{max}^2 - 2s_j^2)$ (Possible gain in energy if T_j is duplicated);
13: **end for**
14: Sort tasks of Q by non-increasing g_i;
15: **for all** task T_j in Q **do**
16: **if** $p_{av} > 0$ **then**
17: $m_j \leftarrow 1$, $p_{av} \leftarrow p_{av} - 1$;
18: **else**
19: $s_j \leftarrow$ the smallest speed that is larger than s_j;
20: **end if**
21: **end for**
22: **end if**
23: **for all** task T_i with $m_i = 0$ **do**
24: Compute the speed s that minimizes $E_i(s)$;
25: **if** $s_i < s$ **then**
26: $s_i \leftarrow s$;
27: **end if**
28: **end for**
29: **return** $< s_i, m_i >$;

Heuristic ThresholdC is based on the same ideas but designed for the case when *continuous* speeds are available. In the first step, tasks speeds are initialized to the speeds which makes $\frac{w_i}{s_i} = P_{\text{t}}$. Then duplicate a specific task to make $P_{\text{nf}} = P_{\text{t}}$. After it, we speed up tasks in L or duplicate them. We proceed similarly as in Threshold, except that instead of choosing the speed which is immediately above the current one, we rather increasing the speed by some parameter Δs.

This parameter should be carefully set: a too small increase of speed Δs will lead to a very small gap between the actual execution time of the task and the bottleneck communication or computation time so that in the event of a task failure, it will take many periods to catch up and no failure should hit the same task during that time.

Heuristic Closer

The previous Threshold or ThresholdC heuristic uses duplication: at least one task is duplicated (in order to fix P_{nf}), which requires spare processors. We propose another heuristic that does not have this requirement. In the Closer heuristic (see Algorithm 14),

after setting all task speeds to the smallest ones so that $\frac{w_i}{s_i} \leq P_{\mathrm{t}}$, we increase the speed of all tasks in L while $P_{\mathrm{exp}} > P_{\mathrm{t}}$ by scaling all tasks simultaneously: we set a coefficient and make sure that for each task, its speed is not smaller than $coef \times s'_i$, where s'_i is the initial speed of T_i. The coefficient is gradually increased until $P_{\mathrm{exp}} \leq P_{\mathrm{t}}$. Finally, we use the same technique as in BestEnergy to attempt to further reduce the energy consumption of tasks.

Algorithm 14 Closer $(n, p, \Delta s)$

1: **for all** tasks T_i **do**
2: $s'_i \leftarrow$ the smallest speed such that $\frac{w_i}{s_i} \leq P_{\mathrm{t}}$, $m_i \leftarrow 0$;
3: **end for**
4: Set $coef \leftarrow 1$;
5: **while** $P_{\mathrm{exp}} > P_{\mathrm{t}}$ **do**
6: $coef \leftarrow coef + \Delta s$;
7: **for all** tasks T_i of set L **do**
8: $s_i \leftarrow$ the smallest speed that is not smaller than $coef \times s'_i$;
9: **end for**
10: **end while**
11: **for all** task T_i **do**
12: Compute the speed s that minimizes $E_i(s)$;
13: **if** $s_i < s$ **then**
14: $s_i \leftarrow s$;
15: **end if**
16: **end for**
17: **return** $< s_i, m_i >$

Heuristic CloserC is the straightforward adaptation of Closer for the case of *continuous* speeds. In a first step, tasks speeds are set as s'_i so that $\frac{w_i}{s'_i} = P_{\mathrm{t}}$. However, in Line 8 of its counterpart, s_i is set to the speed s_i which exactly equals to $coef \times s'_i$ instead of the smallest discrete speed that is not smaller than $coef \times s'_i$.

4.5.3 Bounding the probability of exceeding P_{t}

In this section, we design a heuristic focusing on the constraint on the probability of exceeding P_{t}, thus for the MinEnergy-ProbaC problem.

BestTrade (see Algorithm 15) aims at finding the best tradeoff between energy consumption and the probability of exceeding P_{t}. We consider for each task two critical speeds:

- s_c^i is the speed such that $w_i/s_c^i + w_i/s_{\max} = P_{\mathrm{t}}$; it corresponds to the minimum speed that a task can take without belonging to the S_{excess} set;
- $s_d^i = w_i/P_{\mathrm{t}}$ is the minimum speed that can be assigned to a task: if it is set to a smaller speed, its duration will always exceed the target period.

The idea of the algorithm is first to set all tasks to the smallest speeds that are not smaller than their s_c^i speed. For some tasks, this might be equal to their minimum speed (the smallest possible speed not smaller than s_d^i). In this case, there is no room for reducing speed again without exceeding the target period. For other tasks, we sort them by non-increasing weight: tasks with higher weights contribute the most to the energy dissipation and are thus first slowed down: we select the task T_i with the largest weight, reduce its speed to the minimum possible speed not smaller than w_i/P_{t}. We continue with the tasks of smaller weight, until $P(P_{\mathrm{act}} > P_{\mathrm{t}}) \geq proba_{\mathrm{t}}$. At last, if $P(P_{\mathrm{act}} > P_{\mathrm{t}}) > proba_{\mathrm{t}}$, we just undo the last move to make $P(P_{\mathrm{act}} > P_{\mathrm{t}}) < proba_{\mathrm{t}}$.

We then consider duplication: if duplicating a task T_i (and setting its speed to the smallest speed that is not smaller than s_d^i) is beneficial compared to the current solution (and if a processor is available), the task is duplicated.

Algorithm 15 BESTTRADE (n, p)

```
We assume all weights are different (w_i ≠ w_j for i ≠ j);
for j = 1 to j = n do
    s_j ← smallest possible speed not smaller than w_j/(P_t − w_j/s_max), m_j = 0;
end for
S_reduce ← tasks that have possible speeds between w_j/P_t and w_j/(P_t − w_j/s_max);
sort tasks of S_reduce by non-increasing weight;
k = 1;
while P(P_act > P_t) < proba_t do
    Reduce speed of k-th task in S_reduce to the smallest that is not smaller than w_k/P_t;
    k = k + 1;
end while
if P(P_act > P_t) > proba_t then
    set speed of (k-1)-th task in S_reduce to the smallest that is not smaller than w_k/(P_t −
    w_k/s_max);
end if
p_av ← p − n (Number of available processors);
for j = 1 to j = n do
    s_d ← the smallest speed that is not smaller than w_j/P_t;
    if 2w_j s_d^2 < w_j s_j^2 + f_j(s_j) w_j s_max^2 and p_av > 0 then
        duplicate T_j, m_j = 1, s_j = s_d, p_av ← p_av − 1;
    end if
end for
```

As we did in BESTTRADE, for *continuous* speed model, we still try to keep the best tradeoff between increase in probability and decrease in energy for BESTTRADEC, but now cores can set speed exactly to tasks's critical speeds. First we set all tasks' speed to $s_c^i = w_i/(P_t - w_i/P_t)$. Then, tasks with higher speeds (which are the one with the highest weights) are first slow down: we carefully decrease the speed of all faster tasks to the next critical speed. Whenever the reduction crosses the critical speed s_c^i of some task T_i, this task is included in the set of tasks currently being slowed (S_{excess}). We make sure that no task is assigned a speed smaller than its s_d^i: such tasks are removed from S_{excess} and put into S_{fine}, to remember that their speed cannot be reduced anymore. We stop when the target probability is exceeded: then, all the tasks that were still in S_{excess} are accelerated to reach the exact target probability. Finally, we deal with duplication as in the *discrete* case.

4.6 Experimental validation through simulations

In this section, we evaluate all proposed algorithms through extensive simulations on both real applications and synthetic ones, in the case of *discrete* and *continuous* speeds. For reproducibility purposes, the code is available at github.com/gouchangjiang.

Given a computing platform and an application, we set the target period P_t and probability $proba_t$ so that all assumptions made in the model are true:

- When all tasks are executed with the minimum speed $s_{\min}$, the maximum failure rate is not larger than 10^{-2}. With such a failure rate, the failure of two copies of a duplicated task is very unlikely, and the approximation in Equation (4.2) holds.

Algorithm 16 BESTTRADEC (p, n)

We assume all weights are different ($w_i \neq w_j$ for $i \neq j$).
for $j = 1$ **to** $j = n$ **do**
  $s_j \leftarrow w_j/(P_\mathrm{t} - w_j/s_\mathrm{max})$, $m_j = 0$
end for
$i \leftarrow 0$, $S_{reduce} \leftarrow \emptyset$,
Sort tasks by non-increasing weight, such that $w_1 > w_2 > \ldots > w_n$
while $P(P_\mathrm{act} > P_\mathrm{t}) < proba_\mathrm{t}$ **do**
  $i \leftarrow i + 1$, $S_{fine} \leftarrow \emptyset$
  Put task T_i into S_{reduce}
  $s_c \leftarrow w_{i+1}/(P_\mathrm{t} - w_{i+1}/s_\mathrm{max})$
  $s_d \leftarrow w_i/P_\mathrm{t}$
  for all task T_j in S_{reduce} **do**
    if $\max(s_c, s_d) < w_j/P_\mathrm{t}$ **then**
      Remove T_j from S_{reduce}, put it in S_{fine}
      $s_j \leftarrow w_j/P_\mathrm{t}$
    else
      $s_j \leftarrow \max(s_c, s_d)$
    end if
  end for
end while
Compute s such that if $s_i = s$ for tasks in S_{reduce} and if $s_i = \max(s, w_i/P_\mathrm{t})$ for tasks in S_{fine}, then we have $P(P_\mathrm{act} > P_\mathrm{t}) = proba_\mathrm{t}$
for all task T_i in S_{reduce}, task T_j in S_{fine} **do**
  $s_i \leftarrow s, s_j \leftarrow \max(s, w_j/P_\mathrm{t})$
end for
$p_{av} \leftarrow p - n$; (Number of available processors)
for all task T_i **do**
  if $2w_i(\frac{w_i}{P_\mathrm{t}})^2 < w_i s_i^2 + f_i(s_i) w_i s_\mathrm{max}^2$ and $p_{av} > 0$ **then**
    duplicate T_i: $m_i \leftarrow 1$; $s_i \leftarrow w_i/P_t$; $p_{av} \leftarrow p_{av} - 1$
  end if
end for
return $< s_i, m_i >$ and S_{reduce}

- When all tasks are processed with speed s_max, the maximum failure rate is not larger than 10^{-4}, which means that the failure of a task running at maximum speed is very unlikely.

- P_t should not be smaller than any task duration when running at maximum speed, otherwise, there is no way to meet the target period: $P_\mathrm{t} \geq \frac{\max(w_i)}{s_\mathrm{max}}$.

We set $P_\mathrm{t} = a + \kappa * (b - a)$, where $a = \max(w_i/s_\mathrm{max}, o_i)$ and $b = max(w_i/s_\mathrm{min} + w_i/s_\mathrm{max}, o_i)$: a (respectively b) is the maximal time spent on a task (either on computation or on communication), when running at the maximum (resp. minimum) speed. This way, P_t is never smaller than a, which satisfies the third condition above. Similarly, we avoid the case $P_\mathrm{t} \geq b$, in which the target is too loose, as even the minimum speed can achieve it. A small κ leads to a tighter target period. Under the above three conditions, we set κ to values from 0.05 to 0.95, by increment of 0.01. The target probability is set to $proba_\mathrm{t} = 0.05$ for synthetic applications and $proba_\mathrm{t} = 0.01$ for real applications.

We use the result of heuristic BESTENERGY described in Section 4.4.2 as a comparison basis, as it gives the minimum energy consumption of the system without any constraint.

Possible frequency/voltage	Normalized speed	Failure rate ($\times 10^{-6}/second$)
1.2 Ghz/1.3 V	1	1
987 Mhz/1.16 V	0.80	2.30
744 Mhz/1.03 V	0.61	5.29
502 Mhz/0.89 V	0.41	12.18
260 Mhz/0.75 V	0.21	28.01
66 Mhz/0.675 V	0.055	54.60

Table 4.1 – Configurations of computing platforms.

4.6.1 Multi-core embedded systems

We simulate a multi-core computing platform with 512 cores. Based on AsAP2 and KiloCore, two state-of-art MPSoCs described in Section 4.2, the frequency/voltage options are listed in Table 4.1 [86, 17]. NoC enables extremely fast communications. We describe the value of β together with the output (input) file sizes o_i below in the next subsection. The failure rate is computed as described in Section 4.3.3 as $\lambda(s) = \lambda_0 e^{d\frac{s_{\max}-s}{s_{\max}-s_{\min}}}$. Based on the settings in [98], we set $\lambda_0 = 10^{-6}$ and $d = 4$.

4.6.2 Streaming applications

We use a benchmark proposed in [85] for testing the *StreamIt* compiler. It collects many applications from varied representative domains, such as video processing, audio processing and signal processing. The stream graphs in this benchmark are mostly parametrized, i.e., graphs with different lengths and shapes can be obtained by varying the parameters. Table 4.2 lists some linear chain applications (or application whose major part is a linear chain) from [85]. Some applications, such as time-delay equalization, are more computation intensive than others.

Following the same idea, we also generated synthetic applications in order to test the algorithms on larger applications. We generated 100 groups of linear chains. Each group contains 3,000 linear chains with the same number of nodes, which range from $0.01p$ to p from group to group by an increment $0.01p$, where p is the number of cores. The weights of the nodes w_i follow a truncated normal distribution with mean value 2,000, where the values smaller than 100 or larger than 4,000 are removed. The standard deviation is 500. This ensures that the execution time is not too long so that failure rate is acceptable. The communication time ($\frac{o_i}{\beta}$) follows a truncated normal distribution with mean value $0.001 * P_{\mathrm{t}}$, values that are larger than P_{t} are replaced by P_{t}. Here $P_{\mathrm{t}} = a + 0.05 * (b - a)$.

Application	Size	Average node's weight
CRC encoder	46	14.20
N-point FFT (coarse-grained)	13	1621.31
Frequency hopping radio	16	11815.81
16x oversampler	10	2157.4
Radix sort	13	179.92
Raytracer (rudimentary skeleton)	5	142.8
Time-delay equalization	27	23264.78
Insertion sort	6	475.83

Table 4.2 – Real application examples.

4.6.3 Simulation result

We present both results on synthetic applications and on real applications. On each plot, we show the minimum, mean, and maximum values of each heuristic. In some cases, only the mean is plotted to ease readability, when the minimum and maximum do not bring any meaningful information.

Synthetic applications

Fig. 4.4 presents the results of all heuristics, both in terms of energy consumption, and in terms of constraints, when we vary the parameter κ, hence the tightness of the bound on the expected period. On Fig. 4.4a and Fig. 4.4b, the dashed lines represent the minimum, maximum and average period bound. All chains have $0.5p$ nodes. Apart from MaxSpeed, which always meets the bound, and BestEnergy, which never meets the bound, all heuristics succeed to meet the bound on the expected period. BestTrade, Closer and DuplicateAll are overlapped by Threshold.

Fig. 4.4c and Fig. 4.4d show the probability of exceeding the period bound for *discrete* and *continuous* speed option respectively, and the dashed line is the target $proba_{\mathrm{t}}$. In both cases, DuplicateAll is overlapped by MaxSpeed, and only BestTrade succeeds to always meet the bound. BestEnergy may result in a probability of 0 when $\kappa = 0.95$, but for other values κ, its probability is always 1, which is not depicted in the figure. When *discrete* speed option is selected, Closer and Threshold give very similar results, but sometimes exceed the bound (when $0.4 \leq \kappa \leq 0.8$). Fig. 4.4d also shows that when target period gets larger, it's much easier for all heuristics to meet the probability bound. Since when k gets larger than 0.5, P_{t} is relatively large, so few tasks are in set S_{excess}.

Finally, Fig. 4.4e and Fig. 4.4f depict the energy consumption, normalized by the result of BestEnergy. It does not include the energy cost of heuristic MaxSpeed, which is 215 times larger than BestEnergy. Closer, Threshold and BestTrade are very close to each other in this set of simulations for *discrete* speed option, so some of them are overlapped. The most energy saving heuristic is Closer, then closely followed by BestTrade, then Threshold for *continuous* speed cases. DuplicateAll consumes significantly more energy than the other heuristics. Fig. 4.4f also demonstrates that given a larger period target, our heuristics' performance get close to BestEnergy and the difference between them is smaller in *continuous* speed cases. Overall, both figures show that BestTrade is the best heuristic for these applications: it allows us to always meet both the expected period bound and the probability bound, and it offers similar energy performance as other heuristics. Threshold and Closer are also good options, however they often exceed the probability bound.

Fig. 4.5 illustrates performance of the heuristics for energy consumption and the two constraints, as a function of nodes to cores ratio. Given an amount of cores, a larger ratio corresponds to chains with more nodes. κ is set to 0.4 in these simulations. In Fig. 4.5a and Fig. 4.5b, the dashed lines represent the minimum, maximum and average period bound. On these figures, all heuristics except BestEnergy always meet the target period bound. BestTrade, Closer, DuplicateAll and Threshold overlap, except for their definition domain: DuplicateAll produces a valid allocation only for a ratio of nodes to cores smaller than or equal to 0.5, and Threshold requires to duplicate at least one node. Only BestTrade and Closer are defined for the whole range of the ratio.

Fig. 4.5c and Fig. 4.5d show the probability of exceeding the period bound for *discrete*- and *continuous* speed respectively, where the dashed line is the target probability. Only BestTrade can meet the probability bound for all ratios. In the cases of *discrete* speed, Threshold and Closer give very similar results as BestTrade, except on large ratios (i.e., large chains), where they exceed the bound. In the cases of *continuous* speed, Closer can meet the probability bound when the ratio is small (i.e., small chains). The probability for DuplicateAll and MaxSpeed is always zero.

Finally, Fig. 4.5e and Fig. 4.5f depict the energy consumption as a function of the chain sizes. MaxSpeed is again too large to be included. In the cases of *discrete* speed, the energy cost of BestTrade is the same as Closer and they are close to Threshold. As the size of the chains increases, the energy consumption of other heuristics get close to BestEnergy, and the difference between themselves also get smaller. In the cases of *continuous* speed, except BestEnergy, the most energy saving heuristic is Closer, then closely followed by BestTrade, then Threshold. Once again, Fig. 4.5 shows that BestTrade is the best option for all constraints.

Real applications

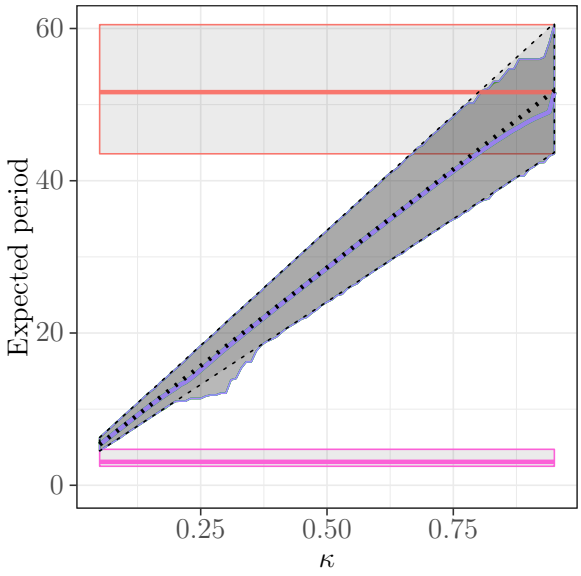
Fig. 4.6 shows the performance of the heuristics on real applications, as a function of κ. In Fig. 4.6a and Fig. 4.6b, the dashed line represents the average period bound. All heuristics except BestEnergy meet the period target. BestTrade, DuplicateAll, Closer and Threshold give very similar results and thus overlap in Fig. 4.6a. BestTrade is partially covered by Threshold and DuplicateAll is covered by Threshold, in Fig. 4.6b.

For probability bound, as shown in Fig. 4.6c and Fig. 4.6d, only BestTrade always meet the target. DuplicateAll and MaxSpeed both give a probability of 0 as before. Closer and Threshold sometimes exceed the target probability by a large factor. Finally Fig. 4.6e and Fig. 4.6f show the energy required by each heuristics. In this setting, BestTrade is the most energy saving heuristic, closely followed by Closer. Threshold requires more energy, and DuplicateAll even more. Not surprisingly, MaxSpeed is the heuristic that costs the most energy, around 254 times larger than BestEnergy (so it is not included in Fig. 4.6e and Fig. 4.6f). This shows that BestTrade is the best heuristic also for real applications.

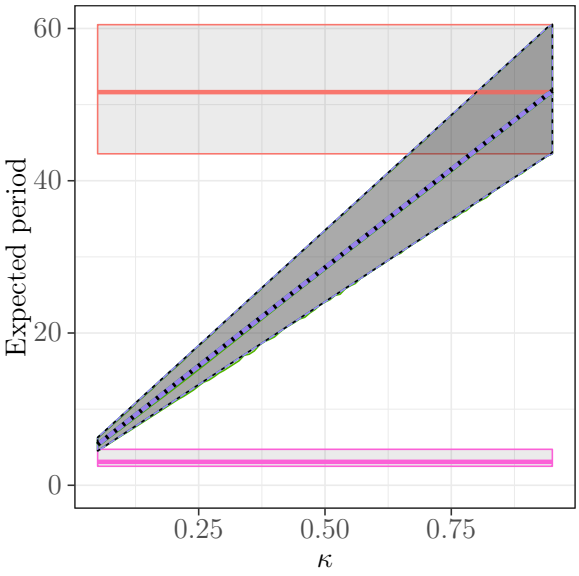
4.7 Chapter summary

In this chapter, we have studied the problem of optimizing the energy consumption of linear chain applications on MPSoCs, which have both reliability and performance constraints. We proposed a new model that allows us to change the frequency of the cores for different tasks and to duplicate some tasks. It takes into account both the expected period, the probability of exceeding the period and the energy efficiency. We proved that minimizing the energy consumption is easy without performance and reliability constraints, but that the problem becomes NP-complete when adding these constraints and when considering a discrete set of possible speeds. We then proposed several heuristics for choosing the tasks' processing speed and which tasks to duplicate. One of the proposed heuristics, BestTrade, is able to meet both bounds on the expected period and on the probability of exceeding the target period, while reducing the energy consumption.

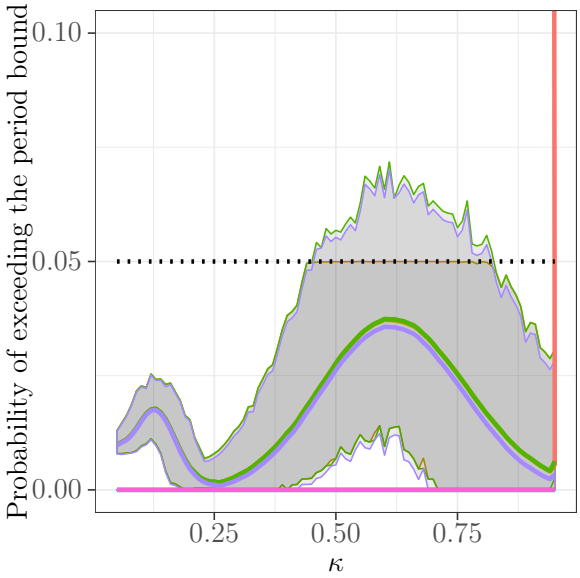
Future work will target more complex allocation schemes, in which several tasks may be mapped on the same core, and in next chapter we will target a more complex task graph. Based on the present results, we expect the problems to become even more complex, but we believe that it will be possible to reuse some ideas derived from the study of linear chains.



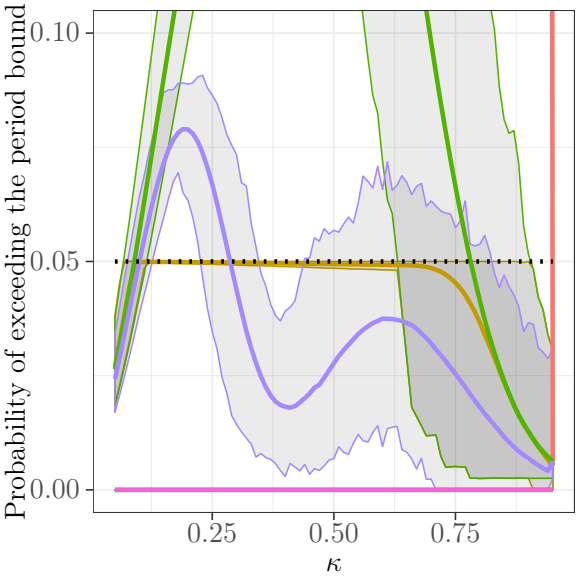
(a) Expected period, *discrete* speed



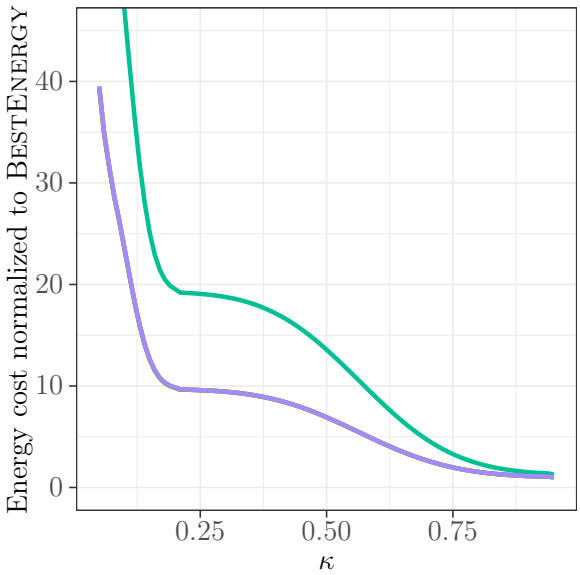
(b) Expected period, *continuous* speed



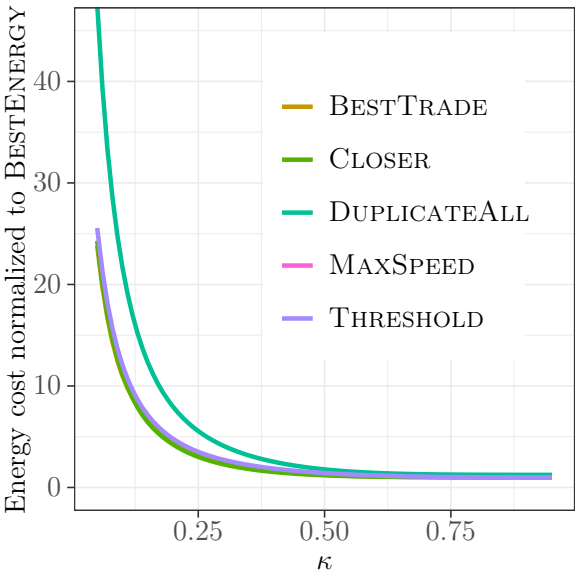
(c) Probability, *discrete* speed



(d) Probability, *continuous* speed

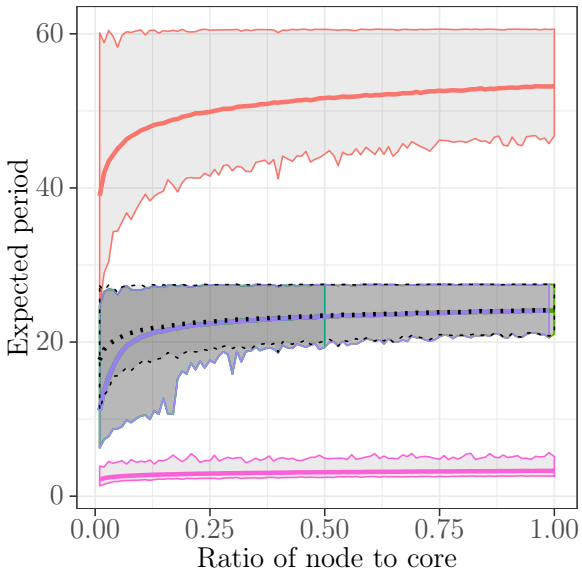


(e) Energy cost, *discrete* speed

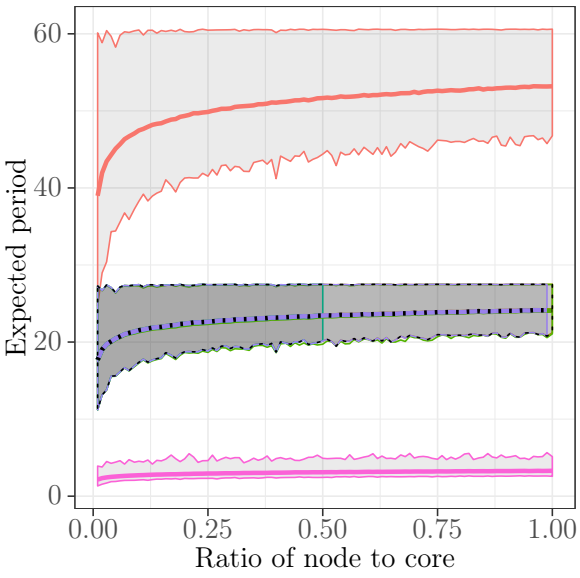


(f) Energy cost, *continuous* speed

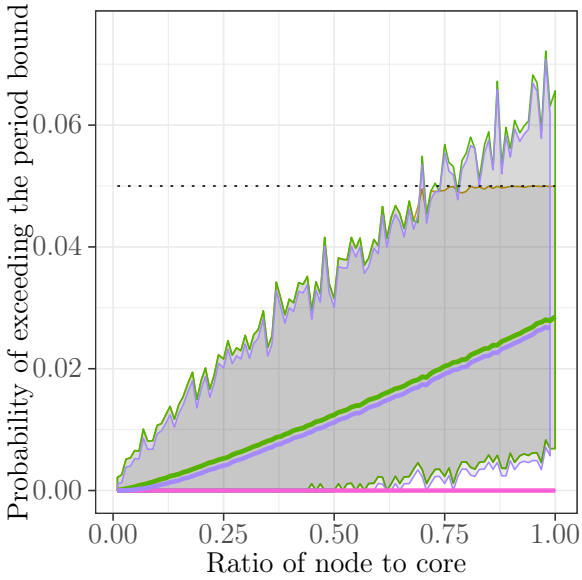
Figure 4.4 – Energy cost and constraints on synthetic applications, as a function of κ.



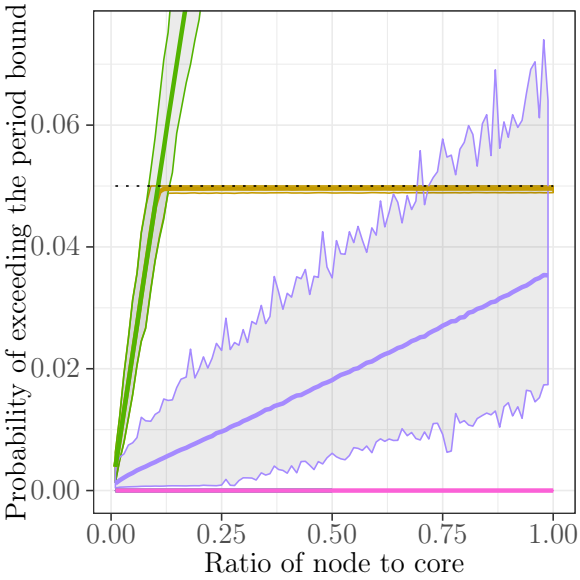
(a) Expected period, *discrete* speed



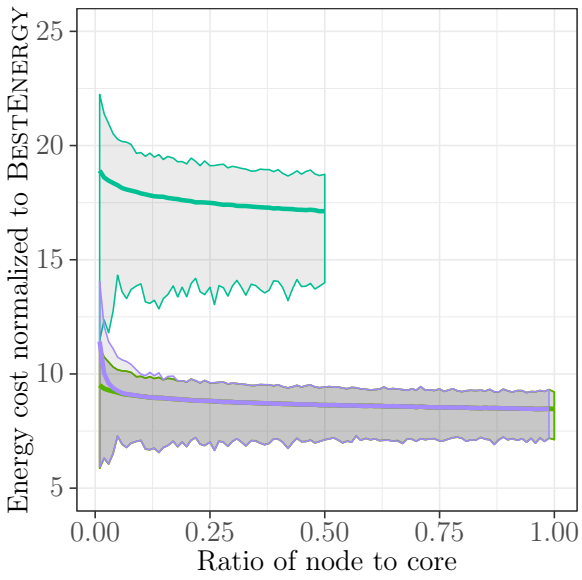
(b) Expected period, *continuous* speed



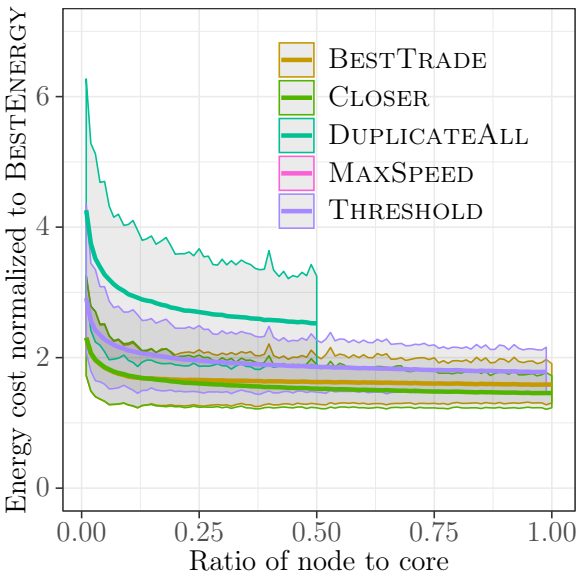
(c) Probability, *discrete* speed



(d) Probability, *continuous* speed

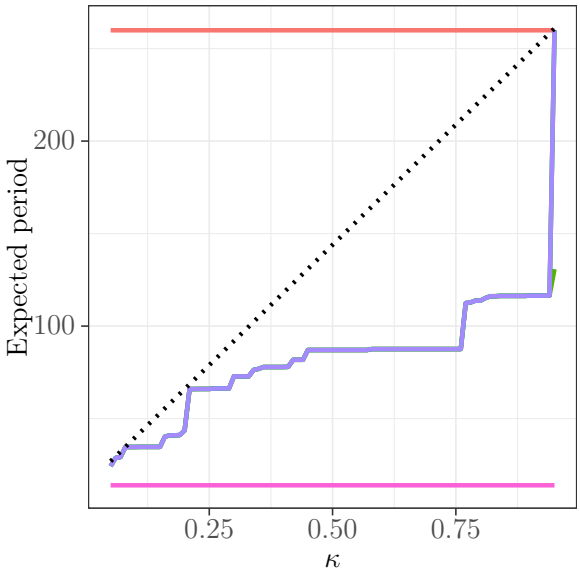


(e) Energy cost, *discrete* speed

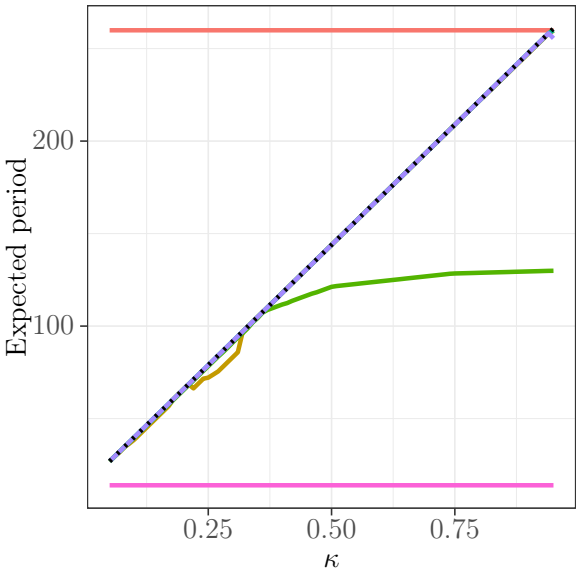


(f) Energy cost, *continuous* speed

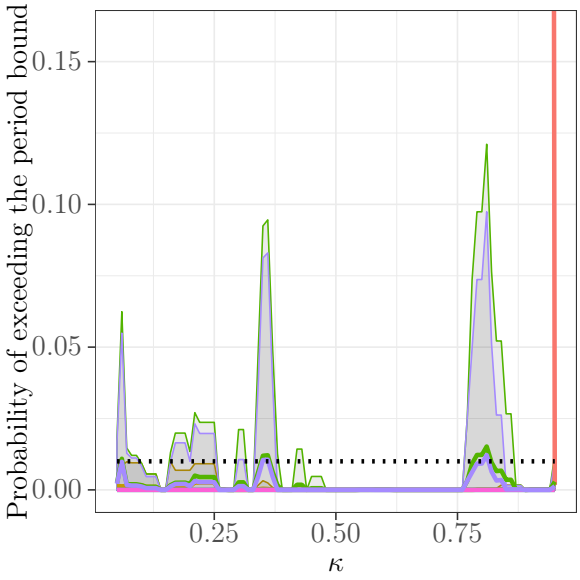
Figure 4.5 – Energy consumption and constraints on synthetic applications, as a function of nodes to cores ratio.



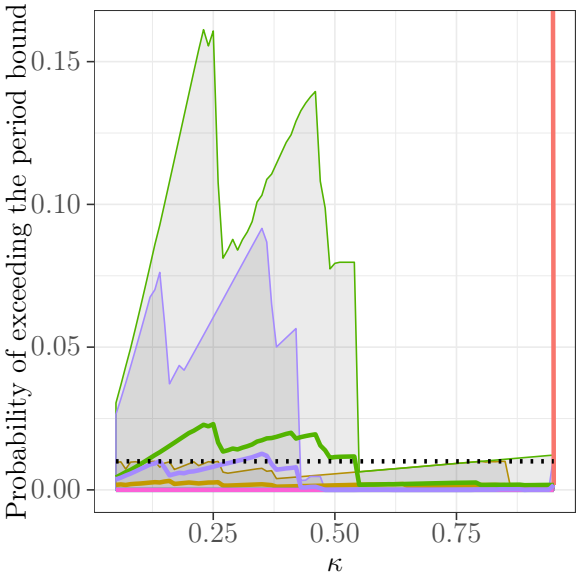
(a) Expected period, *discrete* speed



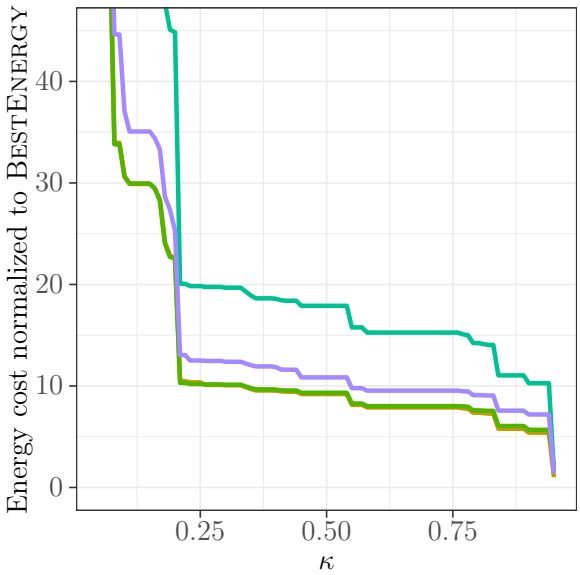
(b) Expected period, *continuous* speed



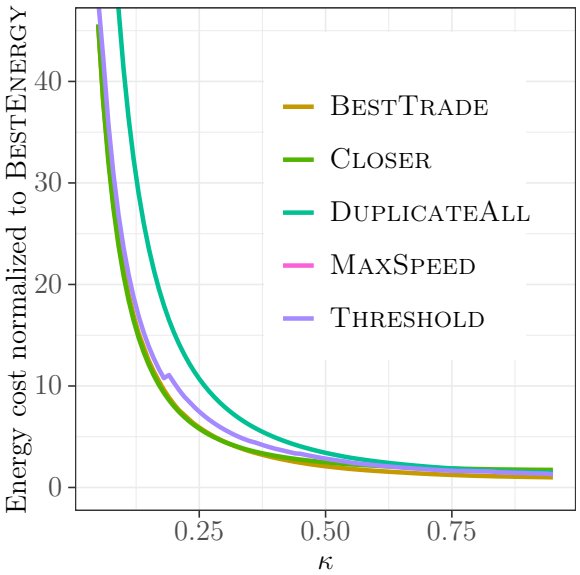
(c) Probability, *discrete* speed



(d) Probability, *continuous* speed



(e) Energy consumption, *discrete* speed



(f) Energy cost, *continuous* speed

Figure 4.6 – Energy consumption and constraints on real applications, as a function of κ.

Chapter 5

Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms

This chapter is a following work of the previous chapter (Chapter 4). The main differences are that the task graphs of streaming applications are more general, and the computing platform we target consists of a two-level communication system, which is very common in embedded systems and in HPC domain. This work has been published at SBAC-PAD 2020 [R1, C1].

5.1 Introduction

Streaming data is continuously generated from applications in high energy physics [32], astronomy [30] and other scientific or industrial domains [26]. With the improvement of detector resolution, it is anticipated that the data volume will dramatically increase. For instance, the advanced light-source facility could generate 1.9 PB data each year and at a rate of 20 GB/sec in the near future [66]. Processing these data in real-time and then feedback key information to decision-making is critically useful, even if it demands intensive computing power. The use of large-scale hierarchical platforms can help parallelize the processing of this streaming data and process it in real time. This may also help reduce the overall energy consumption resulting from the intensive computing properties, for instance by using DVFS.

Most of the workflows corresponding to streaming applications exhibit a regular structure, such as linear chains, trees, fork-join graphs, or general series-parallel graphs. For instance, the majority of the *StreamIt* benchmarks [85] are series-parallel graphs. Hence, we focus on series-parallel applications instead of linear chains. SPGs cover a far larger scope of streaming applications, therefore our model is more general than before. The platform on which we aim at executing such applications is a two-level platform, where several blocks, each with several cores, are available.

To limit the complexity of the problem, the reliability target is relaxed in this study. We consider here a reliability target not equal to 100%, but instead a small percentage of failures is acceptable, so that tasks running at maximum speed have a sufficient reliability. We also observe that triplicating tasks and performing a majority voting leads to a suitable reliability. This avoids overwhelming energy-consuming on applications that do not need an error-free level of reliability.

This chapter makes the following major contributions:

1. We propose a formal reliability and energy-aware model for multi-objective optimization of allocation and scheduling of streaming tasks on a hierarchical platform, and prove that the optimization problem is NP-hard;

2. We design a dynamic programming approach for simple linear chains of streaming tasks, based on which allocation and scheduling heuristics for the general case can be built;

3. Extensive simulations on real applications show that our heuristics can achieve energy savings without degradation of performance and reliability, as compared to running all tasks at the maximum speed.

As a following work, this chapter shares the same related work as Chapter 4, please refer to Section 4.2. The rest of this chapter is organized as follows. Section 5.2 formalizes both application and platform models and defines the MinEnergy optimization problem. Section 5.3 analyzes the problem complexity. Section 5.4 presents a dynamic programming-based solution for MinEnergy when dealing with linear chain applications, and Section 5.5 proposes heuristics for general series-parallel graphs. Section 5.6 evaluates the proposed algorithms. Finally, Section 5.7 concludes the chapter and provides directions for future work.

5.2 Model

5.2.1 Streaming applications – SPGs

The application that is to be scheduled is a streaming application: it operates on a collection of data sets that are executed in a pipelined fashion. The period of the application, which is the inverse of the throughput, corresponds to the time interval between the arrival of two consecutive data sets. We assume that the period of the application (or the throughput) is given by the application and must be enforced. This target period is denoted by P_{t}.

We consider applications represented as a series-parallel graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, or SPG. Nodes of the graph correspond to different application tasks, and are denoted by T_i, with $1 \leq i \leq n$, where $n = |\mathcal{V}|$ is the size of the graph. For each precedence constraint in the application, say from task T_i to task T_j, we have an edge $L_{i,j} \in \mathcal{E}$, and we say that T_j is a successor of T_i, $j \in Succ(i)$. For $1 \leq i \leq n$, w_i is the computation requirement of task T_i, and for each $L_{i,j} \in \mathcal{E}$, with $1 \leq i, j \leq n$, $o_{i,j}$ is the volume of communication to be sent from T_i to T_j before T_j can start its computation.

An SPG is built from a sequence of compositions (parallel or series) of smaller-size SPGs, as illustrated in Figure 5.1. The smallest SPG consists of two nodes connected by an edge. The first node is the source of the SPG while the second is its sink. When composing two SPGs in series, we merge the sink of the first SPG with the source of the second SPG. For a parallel composition, the two sources are merged, as well as the two sinks. The source is also called a *fork* node, and the sink a *join* node.

Data sets arrive at a prescribed rate P_{t}, i.e., a new data set enters the system every P_{t} time units, and we must therefore be able to process at a throughput of at least $\frac{1}{P_{\mathrm{t}}}$. We will further discuss how to compute this processing rate in Section 5.2.6.

5.2.2 Platforms

The computing platform targeted in this work has $c \times p$ homogeneous cores. Each core can run at a different speed, with a corresponding error rate and power consumption. We focus on the most widely used speed model, the *discrete* model, where cores have a discrete number of predefined speeds, which correspond to different voltages at which the core can be operating. Switching is not allowed during the execution of given tasks. The set of speeds is $\{s_{\min} = s_1, s_2, ..., s_k = s_{\max}\}$.

The cores are organized by a hierarchical communication system. It consists in c blocks, each of them containing p computing cores that are tightly coupled by a low-latency interconnect fabric. To have a system with hundreds of cores, blocks are connected by the next

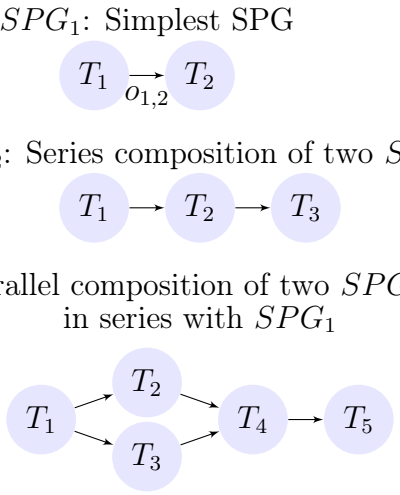


Figure 5.1 – SPG examples.

level network, which contains the route-tables and network parity checking logic. Computation and communication can hence process concurrently. The bandwidth between two cores in the same block and in different blocks are denoted respectively as β_1 and β_2. Communication among cores in the same block is cheaper than that among different blocks [21], i.e., $\beta_1 >> \beta_2$.

5.2.3 Graph partitioning and structure rule

In order to achieve load balance and to save communication, the application is partitioned into several connected parts. Tasks in a part are then allocated to the same core (and a core processes tasks from a single part), hence there is no communication cost to pay between tasks in the same part.

For the ease of the communication pattern, since we consider series-parallel graphs (SPGs), we aim at keeping the SPG structure when creating parts, hence the *structure rule*.

Definition 3 (Structure rule)**.** *A partition of the SPG follows the structure rule if and only if each part consists either of (i) a single task, (ii) a subgraph that is itself an SPG, or (iii) several tasks or SP subgraphs that share the same predecessor and successor (that is, a parallel composition of SP subgraphs).*

If we consider a simple linear chain with three tasks T_1, T_2, T_3, that is, a series composition of these tasks, to be mapped on two cores, this rule does not allow T_1 and T_3 to be mapped on the same core, while T_2 is on another core. Rather, we can either keep the three tasks on one core, or have two consecutive tasks on a core and the third task on another core. For such linear chains, this is similar to *interval mappings* [13].

The rule for parallel compositions is slightly more intricate: consider for instance a simple fork-join with source T_{fork} and sink T_{join} and inner tasks $T_1, \ldots, T_k$, as depicted on Fig. 5.2. Then, either all tasks of this fork-join are in a same part, or T_{fork} and T_{join} must both be in different parts, and none of inner tasks $T_1, \ldots, T_k$ can be in one of these two parts. However, several of them can be grouped in the same part, as they share the same predecessor T_{fork} and the same successor T_{join}. For instance, T_1 and T_3 can be in the same part, while all other tasks $T_2, T_4, \ldots T_k$ can be in another part, as depicted in Fig. 5.2.

A parallel composition of more complex subgraphs is depicted in Figure 5.3 between tasks T_1 and T_{16}. In the proposed partition, two subgraphs of the parallel composition are grouped together (green partition), which is allowed as they share the same predecessor T_1 and successor T_{15}. The other subgraph of this parallel composition is split into two parts. One of them, including T_2 and T_3, is made of two tasks sharing the same predecessor and

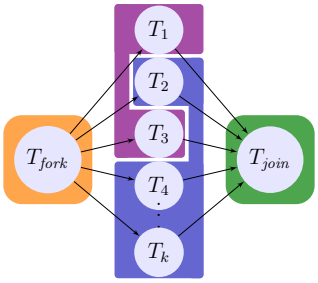


Figure 5.2 – Fork-join graph and a partition following the structure rule.

successor, while the other one is a SP subgraph. Note that by construction, each part of a partition following the structure rule has either a single source vertex and sink vertex (in the cases (i) and (ii) of the definition), or it has a single predecessor and a single successor (case (iii)).

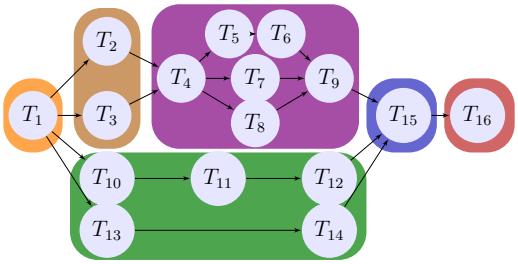


Figure 5.3 – SPG partition following the structure rule.

Notations. The set of task indices that are mapped onto a core v is denoted by C_v, and all these tasks are executing at the same speed $S(v)$. Indices of tasks that are mapped on a block d of cores is denoted as set ℓ_d, and it is the union of the C_v's for all cores v in block d, i.e., $\ell_d = \cup_{v \in d} C_v$.

The sets $Source_v$ (resp. $Sink_v$) represent the indices of the source vertices (resp. sink vertices) mapped on core v. There is only one source and one sink, except for parallel SPGs mapped in a same part. Also, we define the set $PredC_v$ (resp. $SuccC_v$), which contains the core indices on which there are tasks that send outputs to tasks T_i, with $i \in Source_v$ (resp. receive inputs from tasks T_i with $i \in Sink_v$). By construction, either there is only one source and one sink ($|Source_v| = |Sink_v| = 1$), or there is only one predecessor and successor task.

5.2.4 Soft-errors and triplication

High performance computing platforms are subject to failures, and in particular transient errors caused by radiation. In our framework, we can choose the execution speed of a core. However, a very small decrease of speed leads to an exponential increase of failure rate [98, 99]. Indeed, as shown in Chapter 4.3.3, radiation-induced transient failures follow a Poisson distribution, and the fault rate is given by:

$$\lambda(s) = \lambda_0 e^{d \frac{s_{\max} - s}{s_{\max} - s_{\min}}},$$

where $s \in [s_{\min}, s_{\max}]$ denotes the running speed, d is a constant that indicates the sensitivity to DVFS, and λ_0 is the average failure rate at speed $s_{\max}$. λ_0 is usually very small, of the order of 10^{-5} per hour [7]. Therefore, we can assume that the application is reliable enough when running at speed $s_{\max}$, and that there is no need of re-execution [9].

To save energy while having a reliable execution, we also propose a triplication of tasks: three copies of the same task (or group of tasks) are run simultaneously, and a majority voting determines the correct results. Such a scheme may fail only if two copies (among the three) fail simultaneously. For example, on the processor used for the simulation (see

Section 5.6), and when considering that the failure rate at maximum speed is $\lambda_0 = 10^{-5}$ faults per hour, the failure rate at minimum speed is 5.46×10^{-4} per hour. Then, the probability for at least two copies failing is: $3 \times (5.46 \times 10^{-4})^2 = 8.94 \times 10^{-7}$ failures per hour, which is much smaller than the probability at maximum speed. We continue this example below to show that in some cases, triplication succeeds to reduce the energy consumption.

Therefore, after a partition of tasks is done (following the structure rule), in order to have a *reliable* execution, either we execute a whole part on a core at maximum speed without triplication (denoted by $m_i = 1$ for any task T_i in the part), or we triplicate the whole part on three different cores (denoted by $m_i = 3$ for any task T_i in the part). In this later case, the execution speed $S(v)$ used by the three cores is set to the minimum speed such that $S(v)P_{\mathrm{t}} \geq \sum_{i \in C_v} w_i$, so as to minimize the energy cost while respecting period bound. We further enforce that these three cores must be in the same block, since they need to communicate, in particular to do the majority voting and decide which result is correct. Note that if a part is triplicated, the majority voting occurs only for the last task of the part.

5.2.5 Energy

We follow a classical energy model, whose power estimation error in a case study is at most 9.4% on average, see for instance [69]. The energy consumption of executing a data item through all tasks is composed of static part and dynamic part: $E = E_s + E_d$. The static component represents the idle leakage current consumption, which is modeled as $E_s = I_s \times V_s \times P_{\mathrm{t}} \times c_a$, where I_s and V_s denote the leakage current and the minimum possible voltage of a core, and c_a denotes the actual number of cores used, since we assume that other cores can be switched off. Since a data item arrives every P_{t} time units, the static energy is consumed during a time P_{t} for each task, on each of the c_a cores.

For a single execution of task T_i running at speed $s(i)$, the dynamic component E_d^i is related to the operating frequency and voltage, $E_d^i = Cs^3(i) \times \frac{w_i}{s(i)} = Cw_i s^2(i)$, in which C denotes the switching capacitance. The supply voltage is scaled in almost linear fashion with the processing frequency [45]. After taking triplication into consideration, the energy cost of the whole application on one data item is therefore:

$$E = I_s V_s P_{\mathrm{t}} c_a + C \sum_{1 \leq i \leq n} m_i w_i s^2(i),$$

where $m_i = 3$ if T_i is triplicated, otherwise $m_i = 1$. Following up with the previous example, we show that triplicating a task may cost less energy than running it at the maximum speed. We use the values used in Section 5.6: $s_{\min}$ and $s_{\max}$ are 1.2 Ghz and 4 Ghz respectively, static power is 2W, $C = 1$. Assume that the task's weight is 1.2 and the period is 1 second. The energy needed for triplicating it at $s_{\min}$ is $3 * (2 + 1.2 * 1.2^2) = 11.184W$, while running it at $s_{\max}$ requires $2 + 1.2 * 4^2 = 21.2W$.

The energy cost of the communication is not negligible in our model. Within a block, communication among processor cores is realized through a remote memory access. Communication between two cores of different blocks is realized by routers on NoC. For a simple transfer of data on edge $L_{i,j}$, the energy cost can be represented by $E_c(L_{i,j}) = \alpha_{i,j} o_{i,j}$, where $\alpha_{i,j}$ is the energy cost for a unit of data sending. $\alpha_{i,j}$ depends on where tasks are located: if task T_i and T_j are allocated onto the same core, then $\alpha_{i,j} = 0$; $\alpha_{i,j} = \alpha_1 > 0$ if tasks are allocated onto two cores of the same block; otherwise $\alpha_{i,j} = \alpha_2$, and $\alpha_1 < \alpha_2$, see [92] for details.

Also, we must consider the influence of triplication. Given $L_{ij} \in \mathcal{E}$ such that $\alpha_{i,j} \neq 0$, i.e., T_i and T_j are mapped on different cores, the energy cost also depends on whether T_i and T_j are triplicated or not. First, if T_i is triplicated, it does a majority voting before the communication occurs: two outputs from two different cores need to be sent to a core in

the same block, hence the energy cost is $(m_i - 1)\alpha_1 o_{i,j}$ (hence this cost is null if $m_i = 1$). Next, the communication between T_i and T_j must be done one or three times, depending whether T_j is triplicated or not, with a cost $m_j \alpha_{i,j} o_{i,j}$.

In total, the energy cost of the whole application on one data set is:

$$E = I_s V_s P_t c_a + C \sum_{1 \leq i \leq n} m_i w_i s^2(i) + \sum_{L_{i,j} \in \mathcal{E} | \alpha_{i,j} \neq 0} ((m_i - 1)\alpha_1 o_{i,j} + m_j \alpha_{i,j} o_{i,j}).$$

5.2.6 Timing definition and constraints

The actual time spent by tasks mapped on core v is:

$$T(v) = \max \Big(\frac{\sum_{i \in C_v} w_i}{S(v)} + (m_i - 1) \sum_{j \in Sink_v} \sum_{k \in Succ(j)} \frac{o_{j,k}}{\beta_1}, \max_{u \in SuccC_v} \sum_{j \in Sink_v, k \in Source_u} \frac{o_{j,k}}{\beta_{v,u}}, \max_{u \in PredC_v} \sum_{j \in Sink_u, k \in Source_v} \frac{o_{j,k}}{\beta_{u,v}} \Big),$$

where $\beta_{u,v} = \beta_{v,u}$ since communication channels are symmetrical, $\beta_{u,v} = \beta_1$ if u and v are on the same block, otherwise $\beta_{u,v} = \beta_2$. If tasks in C_v are triplicated, then $m_i = 3$, otherwise $m_i = 1$.

The first term in the maximum is the execution time plus the time required for majority voting if tasks are triplicated. Indeed, in this case, two copies of all outputs from task T_j, with $j \in Sink_v$, need to be sent to a core in the same block, since they are sent to the same place. The communication is sequentially executed to avoid potential contention, thus the time needed is two times ($m_i - 1 = 2$ in this case) a single transfer. The second and third terms are the time needed to send and receive datasets.

To execute a data item through all stages of $\mathcal{G}$, the time taken is therefore:

$$T(\mathcal{G}) = \max_{1 \leq v \leq cp} T(v).$$

In order for the mapping to be valid, this has to be less than or equal to the target period

$$T(\mathcal{G}) \leq P_t.$$

5.2.7 Optimization problem

The objective is to minimize the expected energy consumption per dataset of the whole workflow, while ensuring a reliable execution of the application. Hence, each task should either be executed at maximum speed, or triplicated. The goal is hence to decide which tasks to group in a same part, which parts to triplicate, at which frequency to operate each part, and on which core a part should be executed. More formally, the problem is defined as follows:

(MinEnergy) *Given a series-parallel graph composed of n tasks, a computing platform composed of c blocks, each equipped with p homogeneous processor cores that can be operated with a speed within set S, an intra-block (resp. inter-block) communication bandwidth β_1 (resp. β_2, with $\beta_1 >> \beta_2$), and a target period P_t, the goal is to partition the graph and decide, for each part, whether to triplicate it or not, at which speed to operate it, on which core to operate it, so that the total expected energy consumption is minimized, under the constraint that the actual period $T(\mathcal{G})$ should not exceed the period bound P_t (to ensure required performance), and that each task is either executed at maximum speed or triplicated (to ensure reliable execution).*

5.3 Problem complexity

As many partitioning problems, the MinEnergy optimization problem unsurprisingly turns out to be NP-complete. We establish its NP-completeness:

Theorem 4. *The decision version of* MinEnergy *problem is strongly NP-complete.*

Proof. First, we verify that given a mapping of the tasks on the processors, it is possible to verify that (i) each partition follow the structure rule, (ii) the constraint on the execution time is satisfied, and (iii) the required energy of the mapping does not exceed the bound.

To prove the problem NP-hard, we perform a reduction from 3-Partition, which is known to be NP-complete in the strong sense [43]. We consider the following instance $\mathcal{I}_1$ of the 3-Partition problem: let $\{a_1, \ldots, a_{3m}\}$ be $3m$ integers, and B the integer such that $\sum_{i=1}^{3m} a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathcal{I}_1$, we need to solve the following question: does there exist a partition of the a_i's in m subsets $S_1, \ldots, S_m$, each containing exactly 3 elements, such that, for each S_k, $\sum_{i \in S_k} a_i = B$? We build the following instance $\mathcal{I}_2$ of MinEnergy: we consider a fork-join graph as depicted in Figure 5.2, where $w_{fork} = w_{join} = B$, and $w_i = a_i$. The data carried by edges are assumed of negligible size, and thus $o_{i,j} = 0$ for all $i, j \in \mathcal{E}$. We consider a platform with $c = 1$ block of $p = m + 2$ processors, with a set of possible speeds reduced to a single one: $s_{\min} = s_{\max} = 1$. The target period is $P_{\mathrm{t}} = B$. Since we consider the decision version of MinEnergy, we set a bound on the energy: $E \leq I_s \times V_s(m+2)B + C(m+2)B$.

Assume first that there exists a solution to $\mathcal{I}_1$, i.e., that there are m subsets S_k of 3 elements with $\sum_{i \in S_k} a_i = B$. In this case, we build the following mapping as a solution for $\mathcal{I}_2$: T_{fork} and T_{join} are each mapped on a dedicated processor, while for each $1 \leq k \leq m$, the 3 tasks T_i with $i \in S_k$ are mapped on a dedicated processor (no triplication is used). On the whole, the mapping uses $m + 2$ processors. We verify that the computation load of each processor is B, which ensures that both the period bound and the energy bound are met. Besides, this mapping is similar to the one depicted in Figure 5.2 and thus follows the structure rule.

Reciprocally, assume that there exists a solution to problem $\mathcal{I}_2$, that is, a mapping of tasks that respects all bounds as well as the structure rule. We notice that the total computation load of $(m+2)B$ has to be perfectly balanced on the $m+2$ available processors to reach the period bound B, and that no triplication is possible. Hence, T_{fork} and T_{join} (each of computational weight B) must be mapped on dedicated processors, while m processors are available to compute the T_i's. Since $w_i > B/4$, each processor can accommodate at most 3 tasks. For each of these m processors $P_1, \ldots, P_m$, let S_k be the set of the indices of the 3 tasks mapped on P_k. Thanks to the period bound, we know that $\sum_{i \in S_k} a_i \leq B$ and as $\sum_{i=1}^{3m} a_i = mB$, we have $\sum_{i \in S_k} a_i = B$. Hence, the S_k's form a solution of $\mathcal{I}_1$. □

Since the problem is NP-complete, we first address the easier problem of linear chain applications in Section 5.4, before designing heuristics for the general case in Section 5.5.

5.4 Dynamic programming on a linear chain

If the application is a linear chain, we propose a dynamic programming algorithm to solve MinEnergy. According to the structure rule, the linear chain needs to be partitioned into sub-chains, each of them being assigned to one or three distinct cores, depending whether the sub-chain is triplicated or not. We further consider a *contiguous* allocation, where all cores from a same block are assigned to connected sub-chains (forming together a larger chain).

We consider that we have $c^* \leq c$ available blocks, where the $c^* - 1$ first blocks have p cores available, and the last block has $p^* \leq p$ cores available. We express recursively the minimum energy cost of scheduling tasks T_1 to T_i onto the remaining cores. Either all the

tasks form a single part, or we create a part with tasks $T_{j+1}, \ldots, T_i$ and recursively partition the first j tasks.

Initially, we call $E(n, c, p)$, which partitions the whole chain with all blocks and all cores available. The recursion then writes:

$$\begin{aligned} E(i, c^*, p^*) = \min \Big\{ & E_m(1, i, c^*, p^*), \\ & E_t(1, i, c^*, p^*), \\ & \min_{1 \leq j < i} \big\{ E(j, c^*, p^* - 1) + E_c(j, \alpha_1, \beta_1, 1) + E_m(j+1, i, c^*, p^*), \\ & \qquad E(j, c^* - 1, p) + E_c(j, \alpha_2, \beta_2, 1) + E_m(j+1, i, c^*, p^*), \\ & \qquad E(j, c^*, p^* - 3) + E_c(j, \alpha_1, \beta_1, 3) + E_t(j+1, i, c^*, p^*), \\ & \qquad E(j, c^* - 1, p) + E_c(j, \alpha_2, \beta_2, 3) + E_t(j+1, i, c^*, p^*) \big\} \Big\}, \end{aligned} \tag{5.1}$$

where $E_m(i, j, c^*, p^*)$ (resp. $E_t(i, j, c^*, p^*)$) is the energy cost of executing tasks between T_i and T_j included, at the maximum speed (resp. triplicating the tasks) if there are c^* blocks of cores available, the last one having p^* cores available. Also, $E_c(j, \alpha, \beta, m)$ denotes the energy cost of transferring data of size $o_{j,j+1}$ if T_j and T_{j+1} are in different parts: α and β are the energy costs of transferring a unit of data and the bandwidth respectively (their values depend on whether tasks are in a same block or not), and m indicates whether task T_{j+1} is triplicated or not (we pay the communication either three times, or only once).

In the recursive formula $E(i, c^*, p^*)$, we consider all possible situations: either the sub-chain $T_1, \ldots, T_i$ is mapped in a same part, at maximum speed or triplicated (two first lines), or we cut the chain after T_j. In this case, tasks $T_{j+1}, \ldots, T_j$ are in a same part, triplicated or not, and we consider whether there are in the same block as T_j or in a different block, hence resulting in four different cases.

It remains to express E_m, E_t, and E_c. For E_m, we compute the energy cost as described in Section 5.2.5:

$$E_m(i, j, c^*, p^*) = \begin{cases} +\infty & \text{if } \sum_{i \leq k \leq j} w_k > P_{\mathrm{t}} s_{\max} \\ & \text{or } p^* < 1 \text{ or } c^* < 1, \\ I_s V_s P_{\mathrm{t}} + C s_{max}^2 \sum_{i \leq k \leq j} w_k & \text{otherwise.} \end{cases} \tag{5.2}$$

Note that the energy cost is infinite if the period bound is not respected, or if there is no available core ($c^* < 1$ or $p^* < 1$). The expression of E_t relies on s_s, the minimum speed among speeds at which the execution time of tasks between T_i and T_j is not larger than P_{t} (see Section 5.2.4). We add the energy cost of the majority voting within the same block ($2\alpha_1 o_{j,j+1}$), see Section 5.2.5. The period is infinite if there are less than three cores available, or no block left, or if the period bound cannot be matched:

$$E_t(i, j, c^*, p^*) = \begin{cases} +\infty & \text{if } \sum_{i \leq k \leq j} w_k > P_{\mathrm{t}} s_{\max} \\ & \text{or } p^* < 3 \text{ or } c^* < 1, \\ 3(I_s V_s P_{\mathrm{t}} + C s_s^2 \sum_{i \leq k \leq j} w_k) + 2\alpha_1 o_{j,j+1} & \text{otherwise.} \end{cases} \tag{5.3}$$

Finally, for E_c, the energy is infinite if the communication time is larger than the period, otherwise it is computed as indicated in Section 5.2.5:

$$E_c(j, \alpha, \beta, m) = \begin{cases} +\infty & \text{if } o_{j,j+1} > \beta P_{\mathrm{t}}, \\ m \alpha o_{j,j+1} & \text{otherwise.} \end{cases} \tag{5.4}$$

5.4.1 Case studies to show it is not optimal

In this section, we provide an example to show that the method proposed above is not optimal, because of the contiguous assignment of blocks. Consider a platform with $c = 2$

blocks, each with $p = 4$ cores. Each core can run at a speed in set $S = \{1, 2, 4\}$, with the corresponding operating voltage in set $V = \{1, 2, 4\}$. The characteristics of communications on-chip are given by $\alpha_1 = 1,\ \alpha_2 = 2$ (energy cost) and $\beta_1 = 2,\ \beta_2 = 1$ (bandwidth). The static energy cost of a core of a period is $1P_{\mathrm{t}}$ (i.e., $I_s V_s = 1$), constant C is set to 1. The application is a linear chain with four tasks, the task weights of T_1 to T_4 are $\{4, 4, 1, 1\}$ respectively, and the size of all edges are 0.1. The period bound is $P_{\mathrm{t}} = 1$.

The optimal partition and mapping is: break all edges, each task is a part. The first two tasks are running at the maximum speed and are mapped onto two different blocks, each on a core. The third and forth tasks are triplicated, they both run at speed 1 and are mapped onto two different blocks, each task on three cores. T_2 and T_3 are mapped onto the same block. Then, the energy cost is 137.1 (energy cost of running tasks are $64.1, 64.1, 3.9, 3.9$ for T_1 to T_4, and communication energy cost are $0.2, 0.3, 0.6$ between them).

The optimal partition and mapping proposed above is not a contiguous allocation, and hence it will not be considered by the dynamic programming algorithm. Indeed, since triplication of T_4 uses 3 cores, if T_3 is triplicated as well, it has to move to another block, and the core available in the block with T_4 will never be used. Hence, there is no core left for T_1 (indeed, T_2 and T_1 cannot be in a same part without exceeding the period bound). The minimum energy cost by the dynamic algorithm is 148.4, which is larger than 137.1. It is obtained by having T_1 and T_2 in the first block, T_3 and T_4 in the second block, and by triplicating T_4 only.

There are even cases where no contiguous allocation is possible, and hence the dynamic programming algorithm fails at finding a valid mapping. Consider for instance a linear chain application with eight tasks, all have weight 4, edges between T_2 and T_3, T_6 and T_7 have size 1, other edges have size 2. Other configurations are the same as before. Each task should be mapped onto a different core and operated at maximum speed. Tasks between T_3 and T_6 (both included) should be mapped onto the same block, otherwise the communication time between them will exceed the period. Hence, the dynamic programming algorithm cannot find the solution.

5.4.2 Condition for optimality

For tasks from T_1 to T_n, if indices of blocks at which tasks are assigned to are monotonically increasing or decreasing, we call this mapping monotonic. More formally, it is defined below:

Definition 4 (Monotonic mapping)**.** *In a monotonic mapping, for any tasks T_i and T_j with $1 \leq i < j \leq n$ and blocks d and f such that $i \in \ell_d$ and $j \in \ell_f$, then $d \leq f$.*

Lemma 2. *The previous dynamic program producing $E(n, c, p)$ finds a mapping whose energy cost is minimal among monotonic mappings.*

Proof. We prove that for any i, c^*, p^* with $i \in \mathbb{N}$, $c^* \in \mathbb{N}$, $p^* \in \mathbb{N}$, $E(i, c^*, p^*)$ finds a mapping whose energy cost is minimal among monotonic mappings. Then $E(n, c, p)$ is naturally the optimal solution.

We first prove that for an application composed of a single task T_1, solution given by the formula is optimal. $E(1, c^*, p^*) = min(E_m(1, 1, c^*, p^*), E_t(1, 1, c^*, p^*))$. The solution returned is the minimum between the energy cost of running T_1 at the maximum speed on processor p^* of block c^* and that of triplicating T_1 at the speed s_s on processors p^*, $p^* - 1$, $p^* - 2$ of block c^*. These two situations include all possibilities: running T_1 at the maximum speed on a core or triplicating T_1 at speed s_s on three cores. Since cores and blocks are homogeneous, taking any of them costs the same energy. So it takes the minimum of all possibilities, which is apparently the optimal solution.

Then we assume that for applications that has at most k tasks, $k \leq i - 1$, $E(k, c', p')$ returns an optimal monotonic solution, $c' \leq c^*$ and $p' \leq p^*$; then we need to prove that $E(i, c^*, p^*)$ is optimal among monotonic mappings for applications that have i tasks with c^* block and p^* cores on each block.

We consider an optimal monotonic mapping M_{opt}, in which we assume the last edge broken is $L_{j,j+1}$, $(1 \leq j \leq i-1)$. That is to say, tasks from T_{j+1} to T_i are to be mapped onto the same processor. Assume task T_j is assigned to block c_t, $c_t \leq c^*$. Since the mapping is monotonic, tasks between T_{j+1} to T_i can use only block c_t or $c_t + r$, $r = 1, 2, 3, \ldots$. Then we consider all possibilities of mapping.

The energy cost of tasks between T_1 and T_j is denoted by E_{left}.

- if T_{j+1} to T_i are running at the maximum speed on a processor of block c_t, then the energy cost of this part is $E_m(j+1, i, c_t, p'+1)$, the communication energy cost between T_j and T_{j+1} is $E_c(\alpha_1, \beta_1, o_{j,j+1}, 1)$. In total, the energy cost of the application is $E_m(j+1, i, c_t, p'+1) + E_{left} + E_c(\alpha_1, \beta_1, o_{j,j+1}, 1)$;

- if T_{j+1} to T_i are running at the maximum speed on a processor of block $c_t + r$, then the energy cost of this part is $E_m(j+1, i, c_t + r, p^*)$, the communication energy cost between T_j and T_{j+1} is $E_c(\alpha_2, \beta_2, o_{j,j+1}, 1)$. In total, the energy cost of the application is $E_m(j+1, i, c_t + r, p^*) + E_{left} + E_c(\alpha_2, \beta_2, o_{j,j+1}, 1)$;

- if T_{j+1} to T_i are triplicated on processors of block c_t, then the energy cost of this part is $E_t(j+1, i, c_t, p'+3)$, plus the energy cost on communication $E_c(\alpha_1, \beta_1, o_{j,j+1}, 3)$, the total energy cost is $E_t(j+1, i, c_t, p'+3) + E_{left} + E_c(\alpha_1, \beta_1, o_{j,j+1}, 3)$;

- if they are triplicated on processors of block $c_t + r$, the energy cost of this part is then $E_t(j+1, i, c_t + r, p^*)$, the communication energy cost is $E_c(\alpha_2, \beta_2, o_{j,j+1}, 3)$, plus the energy cost of tasks between T_1 and T_j, the total energy cost is $E_t(j+1, i, c_t + r, p^*) + E_{left} + E_c(\alpha_2, \beta_2, o_{j,j+1}, 3)$.

The optimal solution is among them and it costs the least energy. As assumed above, the optimal solution of tasks between T_1 and T_j can be represented as $E(j, c', p')$, $E_{left} \geq E(j, c', p')$. The optimal solution can be rewritten as:

$$\begin{aligned} E_{opt} = \min(&E_m(j+1, i, c_t, p'+1) + E(j, c', p') + E_c(\alpha_1, \beta_1, o_{j,j+1}, 1), \\ &E_m(j+1, i, c_t + r, p^*) + E(j, c', p') + E_c(\alpha_2, \beta_2, o_{j,j+1}, 1), \\ &E_t(j+1, i, c_t, p'+3) + E(j, c', p') + E_c(\alpha_1, \beta_1, o_{j,j+1}, 3), \\ &E_t(j+1, i, c_t + r, p^*) + E(j, c', p') + E_c(\alpha_2, \beta_2, o_{j,j+1}, 3)). \end{aligned}$$

Note that cores and blocks are homogeneous and $c_t \leq c' \leq c^*$, $p' \leq p^*$, hence if we keep the relative place of blocks and cores where tasks are allocated. Then it can be rewritten as:

$$\begin{aligned} E_{opt} = \min(&E_m(j+1, i, c^*, p^*) + E(j, c^*, p^* - 1) + E_c(\alpha_1, \beta_1, o_{j,j+1}, 1), \\ &E_m(j+1, i, c^*, p^*) + E(j, c^* - 1, p^*) + E_c(\alpha_2, \beta_2, o_{j,j+1}, 1), \\ &E_t(j+1, i, c^*, p^*) + E(j, c^*, p^* - 3) + E_c(\alpha_1, \beta_1, o_{j,j+1}, 3), \\ &E_t(j+1, i, c^*, p^*) + E(j, c^* - 1, p^*) + E_c(\alpha_2, \beta_2, o_{j,j+1}, 3)). \end{aligned}$$

Formula $E(i, c^*, p^*)$ considers all situations above, and it further includes running all tasks on one core with the maximum speed or triplicating them on three cores of the same block, and it returns the minimum of all them, so it returns a result that is not larger than the optimal solution, which shows that it is optimal.

□

5.5 Heuristics for series-parallel graphs

For general series-parallel graphs, we first propose a naive baseline heuristic in Section 5.5.1, which will be used to evaluate the performance of the proposed sophisticated heuristics. Other heuristics use a two-step approach to map the SPG onto the platform. The first

step is to partition the graph into many parts, and the second step is to map these parts onto computing resources. We propose two heuristics that focus on partitioning the graph into many parts, and select the most energy efficient way of execution, while the baseline heuristic executes all tasks at maximum speed (Sections 5.5.2 and 5.5.3). The mapping heuristic is described in Section 5.5.4.

5.5.1 Baseline heuristic – MaxS

We first outline a baseline heuristic, MaxS, that will serve as a comparison point. It consists in having each task executed at the maximum speed $s_{\max}$, and then mapping greedily tasks to cores. A set L stores a depth-first traversal of $\mathcal{G}$. At each step, we pop up the first node from L and map it onto current core v until total work load on v, $\sum_{i \in C_v} w_i$, is larger than $P_{\mathrm{t}} s_{\max}$. To respect the *structure rule*, if the node is a fork, we map the whole fork-join onto the current core, otherwise if the workload is already too large, we map the fork onto current core, and its successors onto other cores. We first use all cores of the current block before using cores of the next block.

Algorithm 19 describes this heuristic. We start from the last core p on the last block c, and move to the next core on the same block if it has any, otherwise we move to the core p on next block $c-1$.

Algorithm 17 $NextCore(c^*, p^*)$

```
1: if p* > 1 then
2:    p* ← p* − 1;
3: else if c* > 1 then
4:    c* ← c* − 1, p* ← p;
5: else
6:    return < 0, 0, ∅ >;
7: end if
8: return < c*, p*, ∅ >;
```

Algorithm 18 $MapNodesOn(T_i, T_j, c, v)$

```
1: for all nodes T_k from T_i to T_j do
2:    set m_k ← 1;
3:    put T_k into C_v;
4:    map T_k onto block c and core v;
5: end for
```

5.5.2 Partitioning heuristic – GroupCell

Heuristic GroupCell partitions the graph in a bottom-up way. It first breaks all edges, except (i) edges that have a large communication cost that cannot be done within the period, i.e., $\delta_{i,j} \geq \beta_1 P_{\mathrm{t}}$; and (ii) all edges in a parallel composition when one of the *fork*'s output edges or *join*'s input edges is too large. Indeed, according to the *structure rule*, edges inside this parallel composition should not be broken. For each resulting part, two choices are considered: either running at maximum speed or triplicating, and the most energy efficient choice is selected. Parts stored in vector V_{maxs} are those that are supposed to run at the maximum speed, while other parts that are supposed to be triplicated are in V_{trip}. For two neighbor parts, if they are both in V_{maxs}, merging them will save the energy cost on communication. We hence merge parts in V_{maxs} if they are neighbors and if the merged part fits within the period bound. In this process, we respect the *structure rule*, i.e., the resulted part should be either an SPG or a combination of parallel branches, see Section 5.2.3 for

Algorithm 19 MaxS($\mathcal{G}, c, p, P_{\mathrm{t}}$)

$L \leftarrow$ a depth-first traversal of $\mathcal{G}$;
$b \leftarrow c$; $v \leftarrow p$; $C_v \leftarrow \emptyset$; $T_{mapped} \leftarrow L[1]$;
while L is not empty **do**
  $T_i \leftarrow$ pop up the first element of L;
  if $C_v \neq \emptyset$ **then**
    if T_i is not a successor of T_{mapped} or T_{mapped} is a fork **then**
      $< b, v, C_v > \leftarrow NextCore(b, v, C_v)$;
      if $b < 1$ then return fail;
    end if
  end if
  if T_i is a fork node **then**
    $w \leftarrow$ sum of weight of all nodes of fork-join of T_i;
    if $w + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{\max}$ **then**
      if $w_i + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{\max}$ **then**
        $< b, v, C_v > \leftarrow NextCore(b, v)$;
        if $b < 1$ then return fail;
      end if
      $MapNodesOn(T_i, T_i, b, v)$;
      $T_{mapped} \leftarrow T_i$;
    else
      $T_j \leftarrow$ join of fork-join of T_i;
      $MapNodesOn(T_i, T_j, b, v)$;
      $T_{mapped} \leftarrow T_j$;
      remove nodes of fork-join of T_i from L;
    end if
  else
    if T_i is a join node **then**
      $< b, v, C_v > \leftarrow NextCore(b, v)$;
    end if
    if $w_i + \sum_{j \in C_v} w_j > P_{\mathrm{t}} s_{max}$ **then**
      $< b, v, C_v > \leftarrow NextCore(b, v)$;
    end if
    if $b < 1$ then return fail;
    $MapNodesOn(T_i, T_i, b, v)$;
    $T_{mapped} \leftarrow T_i$;
  end if
end while

details. If the number of processors requested for the whole graph then exceeds the capacity, we merge parts in V_{trip}, starting with the one with largest input edge weight.

This heuristic is described in Algorithm 20.

Algorithm 20 GroupCell($\mathcal{G}, c, p, P_{\mathrm{t}}$)

```
1: parts ← break all edges except the one whose δ_{i,j} > β_1 P_t;
2: V_maxs ← parts in parts for which running at the maximum speed costs less energy than
   triplication;
3: V_trip ← parts\V_maxs;
4: sort V_maxs by an non-increasing order of input edge size;
5: for i = 1 to i = |V_maxs| do
6:    if part V_maxs[i]'s predecessor is also in V_maxs then
7:       if sum of weight of V_maxs[i] and its predecessor ≤ P_t s_max then
8:          restore the broken edge between V_maxs[i] and its predecessor;
9:          replace V_maxs[i] by the combination of V_maxs[i] and its predecessor;
10:      end if
11:   end if
12: end for
13: sort V_trip by an non-increasing order of input edge size;
14: while |V_maxs| + 3|V_trip| > cp do
15:   part ← pop up the first element of V_trip;
16:   merge part into its predecessor;
17: end while
18: for all part in V_trip do
19:   move it into V_maxs if running at the maximum speed costs less energy;
20: end for
21: for all tasks T_i in V_maxs do
22:   set m_i = 1;
23: end for
24: for all tasks T_i in V_trip do
25:   set m_i = 3;
26: end for
```

5.5.3 Partitioning heuristic – BreakFJ-DP

This second partitioning heuristic builds upon the dynamic programming algorithm that was designed for linear chains. It partitions the graph in a top-down way. First, BreakFJ-DP breaks all input edges of *join* nodes and output edges of *fork* nodes so that resulting parts are either linear chains or single nodes. Dynamic programming algorithm from Section 5.4 is then called on each of them with the same number of cores and blocks given as BreakFJ-DP.

Note that on a linear chain application, BreakFJ-DP is similar than calling the dynamic programming algorithm on the whole chain, except that mapping the parts to the cores is not done in the dynamic program but in a second step, using the mapping heuristic.

This heuristic is detailed in Algorithm 21.

5.5.4 Mapping heuristic

Once a partition has been returned by GroupCell or BreakFJ-DP, one still needs to map the parts onto the cores. The mapping heuristic first maps parts that need to communicate a large amount of data on a same block, whenever possible. In a second step, the remaining parts are mapped to the cores following the topology of the graph: a depth-first traversal

Algorithm 21 BreakFJ-DP($\mathcal{G}, c, p, P_{\text{t}}$)

set $L \leftarrow$ all *fork* and *join* nodes of $\mathcal{G}$;
Parts $\leftarrow$ break output edges of *fork* and input edges of *join* in L;
$C \leftarrow \emptyset$; /*edges broken*/
repeat
  part $\leftarrow$ pop up the first element of *Parts*;
  $< i, j > \leftarrow$ source node and sink node of *part*;
  $< E, C_{cur} > \leftarrow DP(i, j, c, p)$;
  if $E == +\infty$ **then**
    return failure;
  end if
  $C \leftarrow C \cup C_{cur}$;
until *Parts* is empty

of the parts is created, and parts are mapped in this order to the available cores. All cores of the current block are used before starting using cores from a new block. Some parts may be merged into its predecessor or its parallel part when there are no available cores.

If two parts connected by an edge with $\delta_{i,j} > \beta_2 P_{\text{t}}$ are mapped onto different processor blocks, the communication time will violate the period bound, then this mapping is invalid. MapRank takes this into consideration and first maps parts that need to transfer data of size $\delta_{i,j} > \beta_2 P_{\text{t}}$ onto the same block. A part may have more than one edge with $\delta_{i,j} > \beta_2 P_{\text{t}}$, so parts connected by these edges should all be mapped onto the same block. They are grouped into the same vector in the first for loop of MapRank. These vectors are stored in set L. If number of processors needed by a group exceeds the capacity p, we select the part with the smallest computation weight and execute it at the maximum speed on one processor. This process is repeated until the requirement fits the capacity. According to their demand, parts are sequentially assigned to processors $|Sets[b_{cur}]|$, $|Sets[b_{cur}]| + 1$ and so on. MapTopology is called afterwards to map the remaining parts.

In MapTopology (see Algorithm 23), a part is at first mapped onto the same block as its first predecessor, if it is possible. Otherwise, it will be mapped onto the block with the closest index that has enough cores. If the input edge is too large to communicate between two blocks, and current block does not have enough cores, MapTopology first tries to move some parts already mapped onto the current block to the next block, and then continues the mapping from the next block. If it does not work, MapTopology then merges linear chains already mapped from the smallest size until there is enough space.

5.6 Experimental evaluation of the heuristics

In this section, we evaluate all proposed algorithms through extensive simulations on real applications. For reproducibility purposes, the code is available at github.com/gouchangjiang/Stream_HPC.

Algorithm 22 MapRank($\mathcal{G}, C$)

1: $b_{cur} \leftarrow c$; $L \leftarrow \emptyset$;
2: construct quotient graph Q by breaking edges in C;
3: initialize $Sets$ with c empty vectors;
4: $C' \leftarrow$ edges of Q whose $\delta_{i,j} > \beta_2 P_{\mathrm{t}}$;
5: **for** $i = 1$ to $i = |C'|$ **do**
6: $< T_p, T_s > \leftarrow$ nodes connected by edge $C'[i]$;
7: **if** T_p is in L but not T_s **then**
8: push T_s into the same vector as T_p;
9: **end if**
10: **if** T_s is in L but not T_p **then**
11: push T_p into the same vector as T_s;
12: **end if**
13: **if** neither T_p nor T_s is in L **then**
14: initialize a vector with them, push it into L;
15: **end if**
16: **end for**
17: **while** $L \neq \emptyset$ **do**
18: $vec \leftarrow$ pop up the first vector of L;
19: $nbr \leftarrow \sum_{i \in vec} m_i$; /*cores requested*/
20: **if** $nbr > p$ **then**
21: sort vec by an non-decreasing order of weight;
22: $idx \leftarrow 1$;
23: **while** $nbr > p$ **do**
24: set the node $vec[idx]$ to run on only one core;
25: $idx \leftarrow idx + 1$; recaculate nbr;
26: if $idx > |vec|$ then return failure;
27: **end while**
28: **end if**
29: if p - $|Sets[b_{cur}]| < nbr$ then $b_{cur} \leftarrow b_{cur} - 1$;
30: **for** $i = 1$ to $i = |vec|$ **do**
31: **if** node $vec[i]$ needs 3 processors **then**
32: push it into vector $Sets[b_{cur}]$ three times;
33: **else**
34: push it into vector $Sets[b_{cur}]$;
35: **end if**
36: **end for**
37: **end while**
38: MapTopology($Q, Sets$);

Algorithm 23 MAPTOPOLOGY($Q, Sets$)

1: $L \leftarrow$ a depth-first traversal of Q;
2: **repeat**
3: $T_i \leftarrow$ pop up L; $b \leftarrow c$;
4: **if** T_i has not been mapped yet **then**
5: **if** T_i has a predecessor **then**
6: $b_{cur} \leftarrow$ which block the first predecessor of T_i mapped onto;
7: **end if**
8: **if** $p - |Sets[b]| < m_i$ and the size of first input edge is larger than $\beta_2 P_{\mathrm{t}}$ **then**
9: $h \leftarrow$ the index such that all input edges of $Sets[b][h]$ are not larger than $\beta_2 P_{\mathrm{t}}$;
10: **if** h exists and $b > 1$ **then**
11: move elements between $Sets[b][h]$ and the end of $Sets[b]$ to $Sets[b-1]$;
12: $b \leftarrow b - 1$;
13: **else**
14: **while** $p - |Sets[b]| > m_i$ **do**
15: $part_j \leftarrow$ the smallest part of $Sets[b]$ who has only one predecessor that is not a fork;
16: merge $part_j$ to its predecessor and remove $part_j$ from $Sets[b]$;
17: **end while**
18: **end if**
19: **end if**
20: **while** $p - Sets[b] \geq m_i$ and $b > 1$ **do**
21: $b \leftarrow b - 1$;
22: **end while**
23: **if** $m_i == 3$ and $p - |Sets[b]| \geq 3$ **then**
24: put $part_i$ into $Sets[b]$ three times;
25: **else**
26: put $part_i$ into $Sets[b]$;
27: **end if**
28: **end if**
29: **until** L is empty

5.6.1 Simulation setup

ID	Nbr Nodes	Nbr Edges	Max Degree	Max Node Weight	Min Node Weight	Max Edge Size	Min Edge Size
B10	80	95	2	9088	8	136	1
B11	22	38	17	272	96	257	1
B13D	66	80	2	240	12	16	2
B13G	214	313	32	36771	3	1344	1
B14	86	108	4	128	12	16	2
B15	24	38	8	4864	384	64	8
B16	197	229	2	1024	192	96	32
B20	283	469	32	4035	4	128	4
B22	50	62	2	448	192	2	1
B25	43	54	6	1434	8	60	1
B27	85	100	8	11312	6	64	1
B2	132	177	12	44928	6	1728	1
B36	97	128	8	128	12	8	1
B37	110	140	4	128	12	8	1
B38	24	38	8	4864	384	8	1
B3	20	34	15	140	22	15	1
B40	22	36	8	138	138	8	1
B44	46	55	2	29	6	2	1
B45	43	63	9	27648	72	468	12
B46	54	93	12	3528	72	2592	9
B47	31	38	2	208	96	16	2
B49	180	295	32	414144	96	9216	16
B4	12	19	8	579	48	32	1
B50	165	211	32	3274750	259	140	0
B53	16	19	4	181500	24	3300	0
B56	53	67	12	5076	332	12	0
B61	44	45	2	6541490000	3	167316	1
B63	234	267	2	3336	68	256	4
B65	12	15	4	3306	8	4	1
B66	6	7	2	10	6	2	1
B67	116	151	15	9105	6	60	1
B6	170	240	8	126	14	16	2
B7	152	201	8	128	6	16	1
B9	57	73	16	65055	251	1	0

Table 5.1 – Set of streaming apps: general SPGs.

We use a benchmark proposed in [85] for testing the *StreamIt* compiler. It collects many applications from varied representative domains, such as video processing, audio processing and signal processing. 44 applications are selected, in which 10 of them are chains, more details can be found in table 5.1 and table 5.2.

We base our platform parameters on the characteristics of the Intel Skylake-SP Processor [78]: the possible core frequencies are $\{s_{\min} = 1.2, 2.1, 2.4, 2.6, 3.0, 3.7 = s_{\max}\}$, and the idle power of each core is 2.17W. To simulate applications with various communication to computation ratio (CCR), we choose three values of β_1, leading to a CCR (defined as the total time spent on communications over the total time spent on computations) of 10^{-4}, 10^{-3}, or 10^{-2}, while $\beta_2 = \beta_1/16$. α_1 and α_2 are set as 0.2 and 0.8 respectively. C in section 5.2.5

ID	Nbr Nodes	Nbr Edges	Max Degree	Max Node Weight	Min Node Weight	Max Edge Size	Min Edge Size
B19	13	13	1	2464	632	128	128
B39	4	4	1	1576	1104	1	1
B41	6	6	1	745	96	1	1
B54	10	10	1	11360	11	16	1
B57	13	13	1	208	96	1	1
B58	5	5	1	19836	32	2	0
B5	6	6	1	265	96	16	16
B60	5	5	1	473	8	1	1
B64	29	29	1	36960	12840	1920	1080
B8	18	18	1	23	6	1	1

Table 5.2 – Set of streaming apps: linear chains.

is set as 1.

For each application, we set the period bound $P_t = a + (b - a)/\kappa$. The value of a is set to the minimum time spent on a task or a data transfer at speed β_1 ($a = \max(w_i/s_{\max}, \min(\delta_{i,j}/\beta_1))$), which corresponds to a very tight period bound. On the contrary, b is set to the time needed to process all tasks on a core at the minimum speed ($b = \sum_{1 \leq i \leq n} w_i/s_{\min}$), corresponding to a very loose period bound. We set κ to values from 2 to 10, by increments of 2. Note that it may happen that an application cannot meet the period bound, for instance if an edge between two tasks and the sum of computation cost of these tasks both cannot fit within P_t: in that case, all heuristics will fail to produce an appropriate mapping.

Since some heuristics mail fail to produce an acceptable mapping, for each plot described below, we select a subset of applications on which all considered heuristics succeed to produce a mapping, and we plot the average result of the heuristics on this common subset.

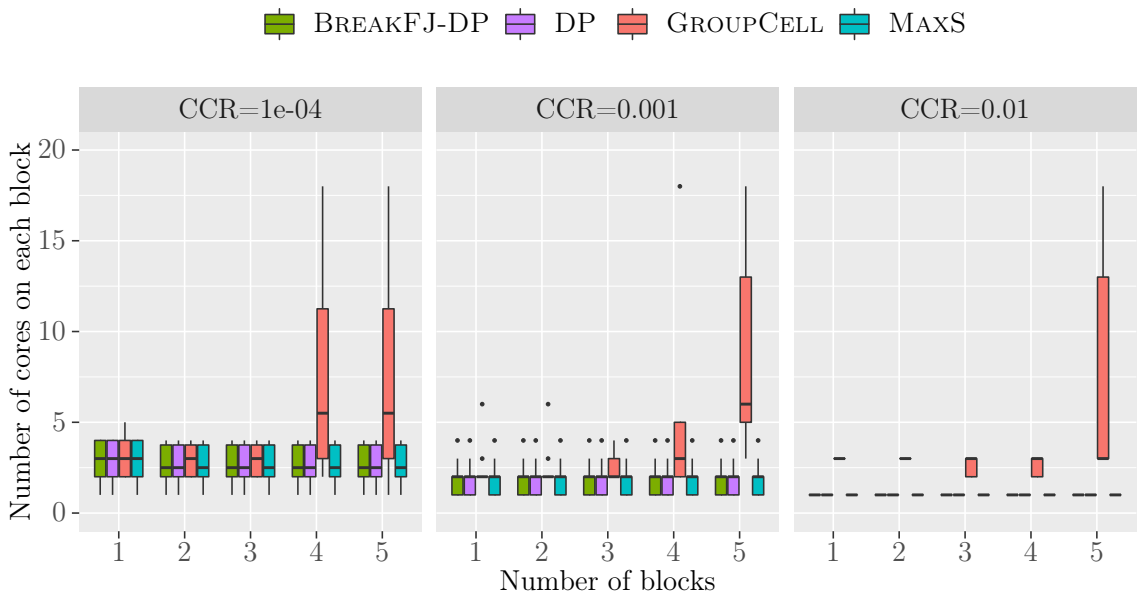
5.6.2 Simulation results

Minimum number of cores request

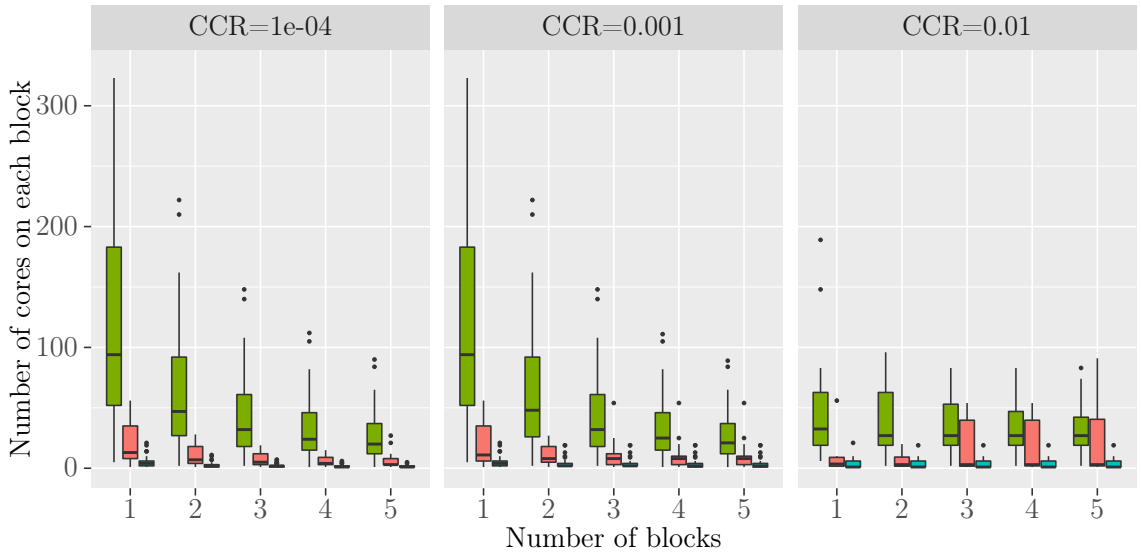
After removing the cases where heuristics do not find a valid solution under a given number of blocks and cores, Fig. 5.4 shows the minimum number of cores on a block requested by each heuristic, with various number of blocks provided, κ is set as 4, a median value. On linear chains, as shown in Fig. 5.4a, dynamic programming (denoted as DP), MaxS as well as BreakFJ-DP have the same performance, they require the least number of cores. GroupCell requires averagely 2.4 times more cores than DP. On general SPGs, as shown in Fig. 5.4b, BreakFJ-DP uses far more cores than GroupCell and MaxS. For instance, with CCR=10^{-3}, BreakFJ-DP uses 5 times more cores than GroupCell averagely.

Energy cost

Fig. 5.5 and Fig. 5.6 depict the energy cost as a function of κ, where a larger κ represents a tighter period, with different number of blocks and cores given. For linear chains, see Fig. 5.5, when 2 blocks and 2 cores on each block are given, 2, 3 and 1 applications are considered for CCR=10^{-4}, 10^{-3} and 10^{-2} respectivley, since at least one heuristic fail to get a valid mapping on other applications. The number of applications included are 10, 8, 9 for CCR=10^{-4}, 10^{-3} and 10^{-2} respectivley, when 4 cores given on each block. BreakFJ-DP and DP reduce by 33% on average compared to MaxS, and around 60% when communications are expensive. It leads to the same conclusion when 2 blocks, each with 8 cores are provided. BreakFJ-DP and DP reduce the energy by 44% on average compared to

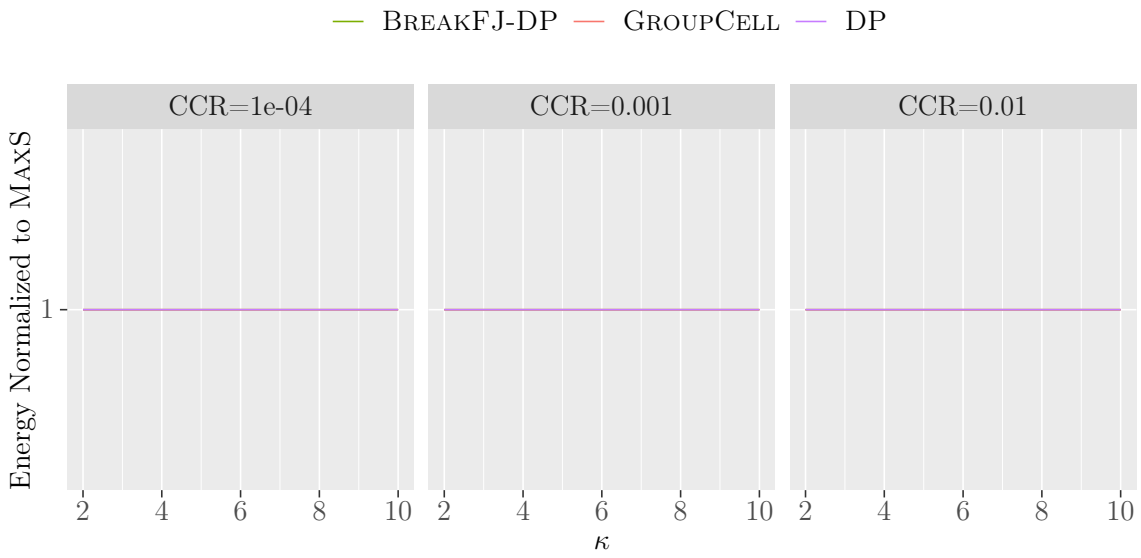


(a) linear chains

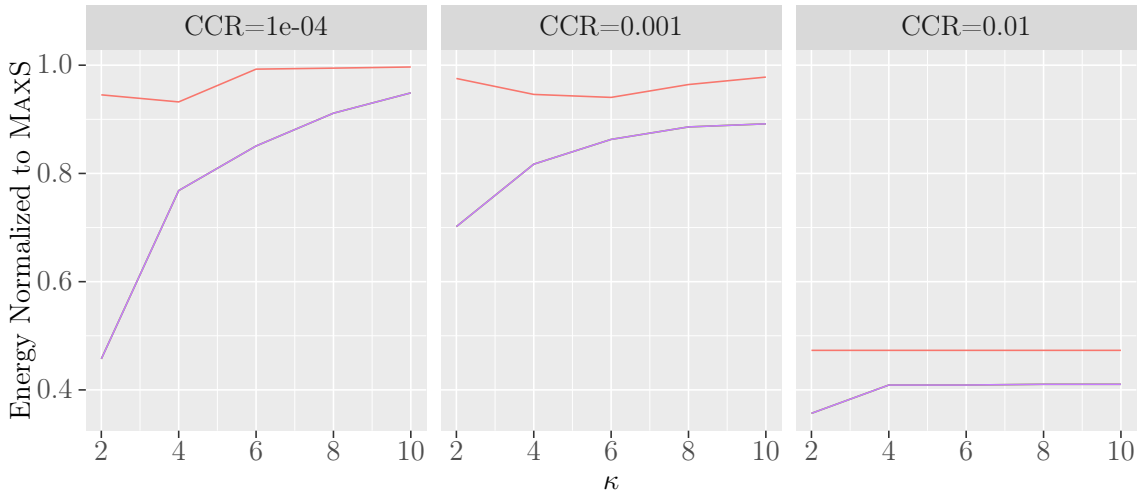


(b) general SPGs

Figure 5.4 – Number of cores requested by each block with different CCR and number of blocks provided. Note that DP is only applicable on linear chains, i.e., *top* figure.



(a) BreakFJ-DP, GroupCell and DP give the same results, hence only DP is visible, 2 cores on each block.



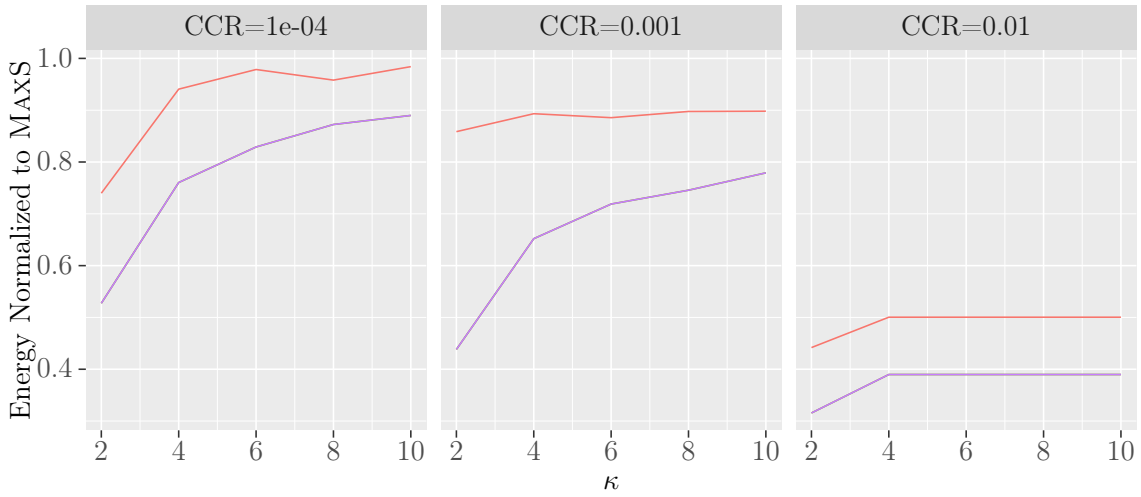
(b) BreakFJ-DP and DP give the same results, hence only DP is visible, 4 cores on each block.

MaxS. Note that when CCR=10^{-4}, the results only include 3 applications out of 10, since GroupCell fails on other applications because of shortage of cores. For CCR=10^{-3} and 10^{-2}, 9 applications are included.

The gains are also very impressive for general SPGs, see Fig. 5.6. When 2 blocks, each with 128 cores given, both heuristics save more than 50% of energy in all settings, with BreakFJ-DP being better for tighter periods and larger CCRs. Note however that the results for CCR=10^{-2} are computed only on a small subset of applications, 6 out of 34, since the heuristics failed on the other applications: the period bound could not be met because of the high communication cost on some edges. For CCR=10^{-3} and 10^{-4}, 31 and 32 applications are included respectively.

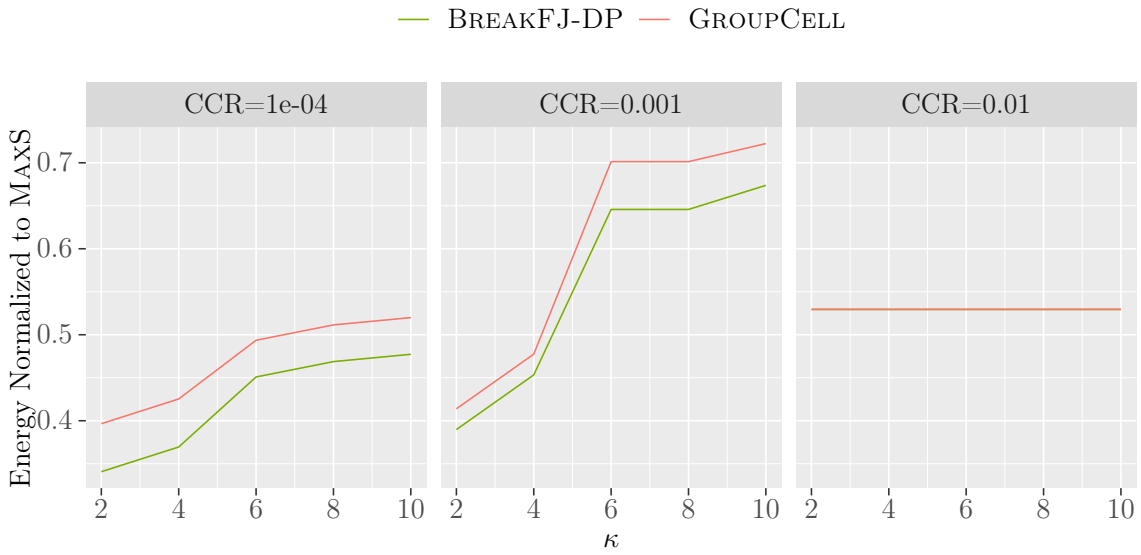
With 2 blocks, each has 64 cores equipped, both heuristics also save more than 50% of energy in all settings, with BreakFJ-DP being better at least 5% in most cases. 27, 20 and 4 applications are included for CCR = 10^{-4}, 10^{-3} and 10^{-2} respectively, since at least one heuristic does not return a valid mapping on other applications.

With 2 blocks, each has 32 cores equipped, BreakFJ-DP and GroupCell still outweigh MaxS by around 44% when CCR=10^{-4}.BreakFJ-DP is still slightly better than GroupCell. 17, 11 and 2 applications are included when CCR=10^{-4}, 10^{-3} and 10^{-2} respectively.

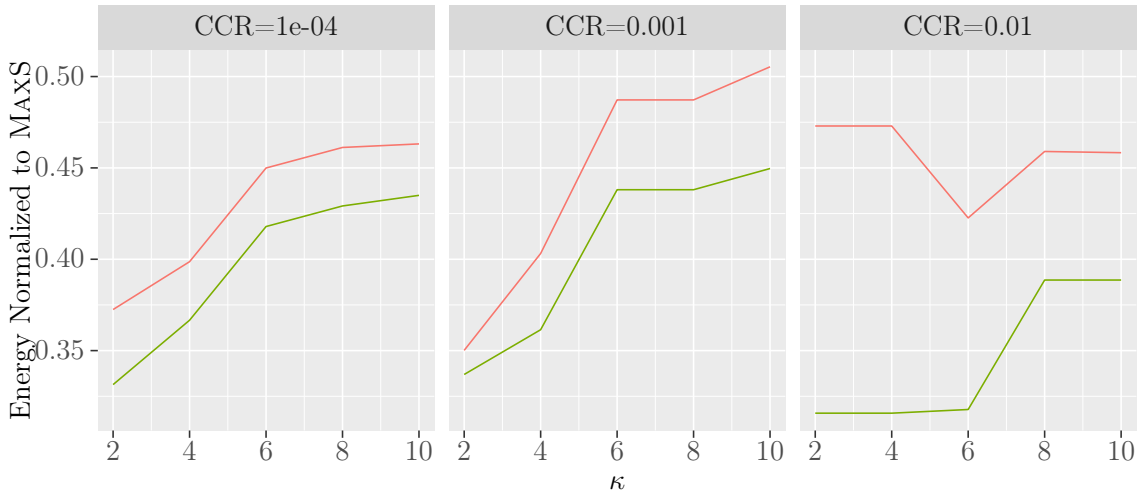


(c) BreakFJ-DP and DP give the same results, hence only DP is visible, 8 cores on each block.

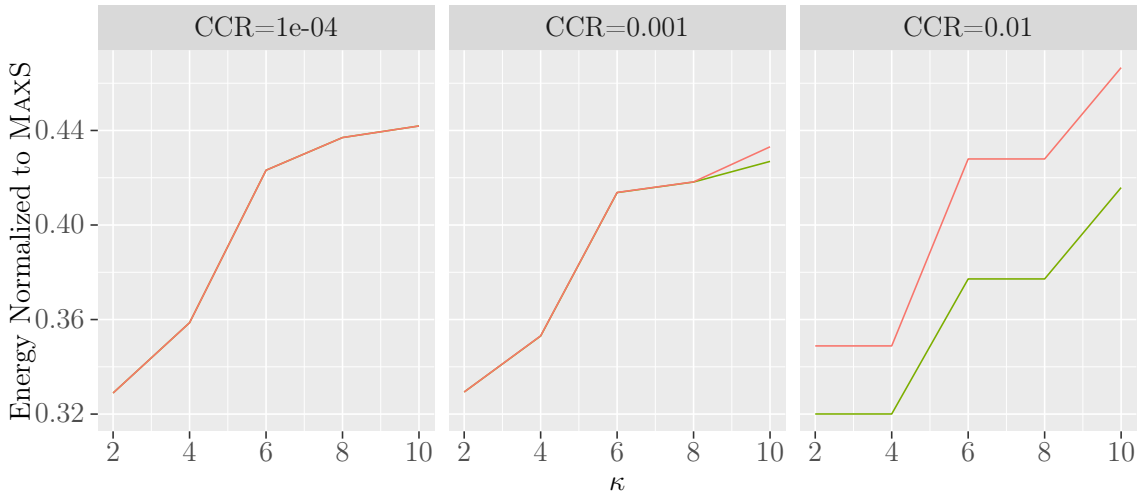
Figure 5.5 – Energy consumption on linear chains relative to MaxS as a function of the period bound tightness κ.



(a) For $CCR = 10^{-2}$, both heuristics give the same results, 4 blocks, each with 32 cores provided.



(b) 4 blocks, each with 64 cores provided.



(c) For $CCR = 10^{-4}$, both heuristics give the same results, 4 blocks, each with 128 cores provided.

Figure 5.6 – Energy consumption on general SPGs relative to MaxS as a function of the period bound tightness κ.

Failure cases

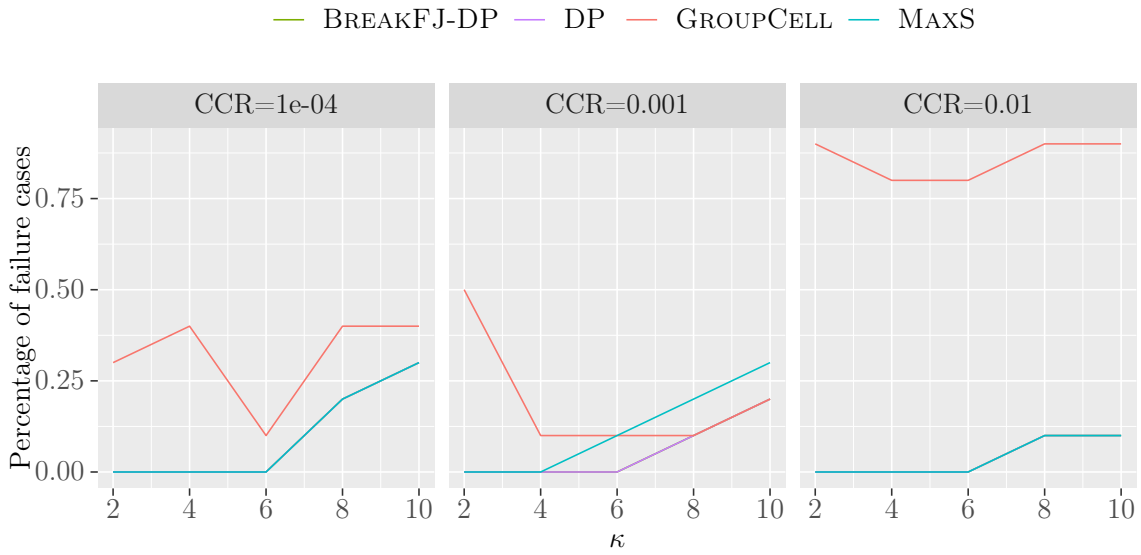
We report percentage of failure cases in Fig. 5.7 and Fig. 5.8. Note that in total there are 34 general SPGs and 10 linear chains. On linear chains, GroupCell fails more than other heuristics. GroupCell, DP has the same number of failure cases (around 10% when each block has 4 or 8 cores.), they are covered by MaxS as MaxS has a slightly more failure cases than them.

Percentage of failure cases on general SPGs are shown in Fig. 5.8. BreakFJ-DP is the one who fails more than other heuristics since it requests more cores. As one can see that with a larger number of cores on each block and communication is not so expensive (CCR=10^{-4} or 10^{-3}), percentage of failures cases of BreakFJ-DP decrease from 0.4 to 0.2 and then to zero. The same for other heuristics, more cores on a block, on less applications they fail. With a tight communication bandwidth (CCR=10^{-2}) and a tight period bound ($8 \leq \kappa \leq 10$), more than 50% fail.

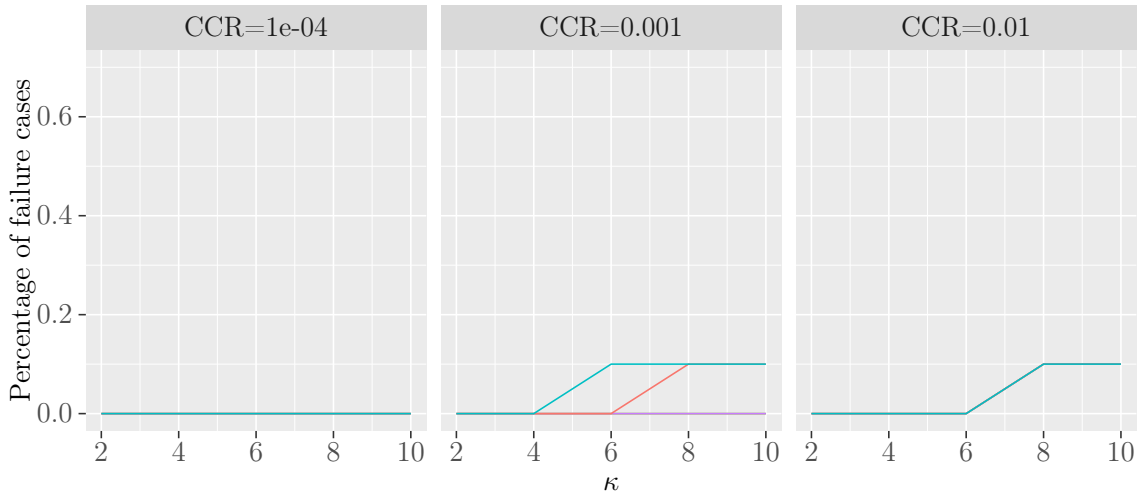
5.7 Chapter summary

We have addressed the problem of mapping streaming SPG applications onto a hierarchical two-level platform, with the goal of minimizing the energy consumption, while ensuring performance (a period bound should not be exceeded) and a reliable execution (each task should either be executed at maximum speed or triplicated). We have formalized the problem and proven its NP-completeness, and provided practical solutions building upon a dynamic programming algorithm, which returns the optimal *contiguous* mapping for a linear chain. Heuristics are proposed for general SPGs, and the BreakFJ-DP heuristic that builds upon the DP algorithm provides significant savings in terms of energy consumption, with more than 61% savings, in particular when the period bound is not too tight. With tighter period bounds, we still achieve 57% savings.

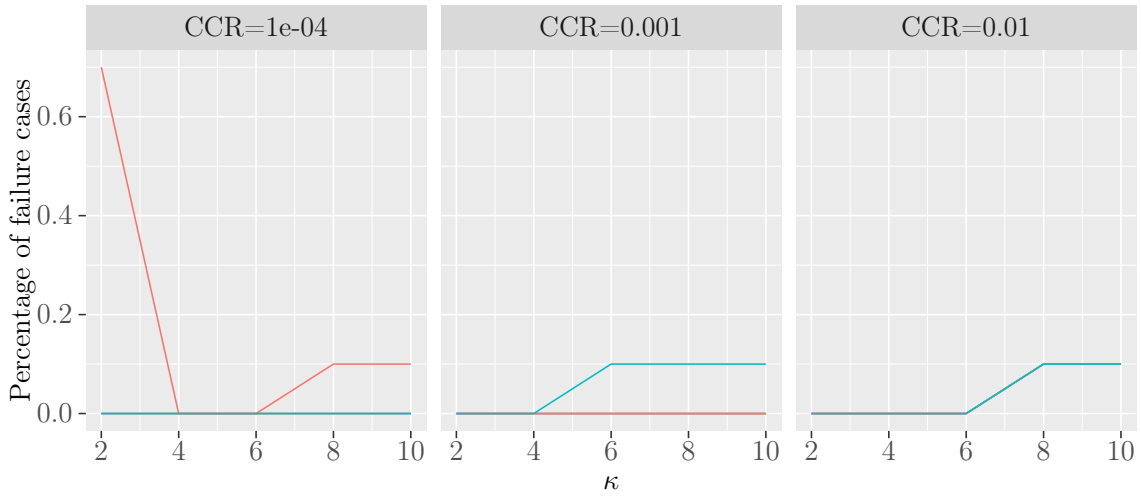
However, this heuristic may fail with limited number of cores per blocks. In this case, our GroupCell heuristic is an interesting alternative, with only a slightly greater energy consumption for a reduced number of cores used. An interesting open question is whether the proposed dynamic program is an approximation algorithm: even though it is not optimal in the general case, it works well in practice and it would be interesting to provide a guarantee on its performance.



(a) Linear chains. 2 blocks, each with 2 cores provided. BreakFJ-DP, DP have the same number of failures as MaxS, hence they are covered by MaxS sometimes.

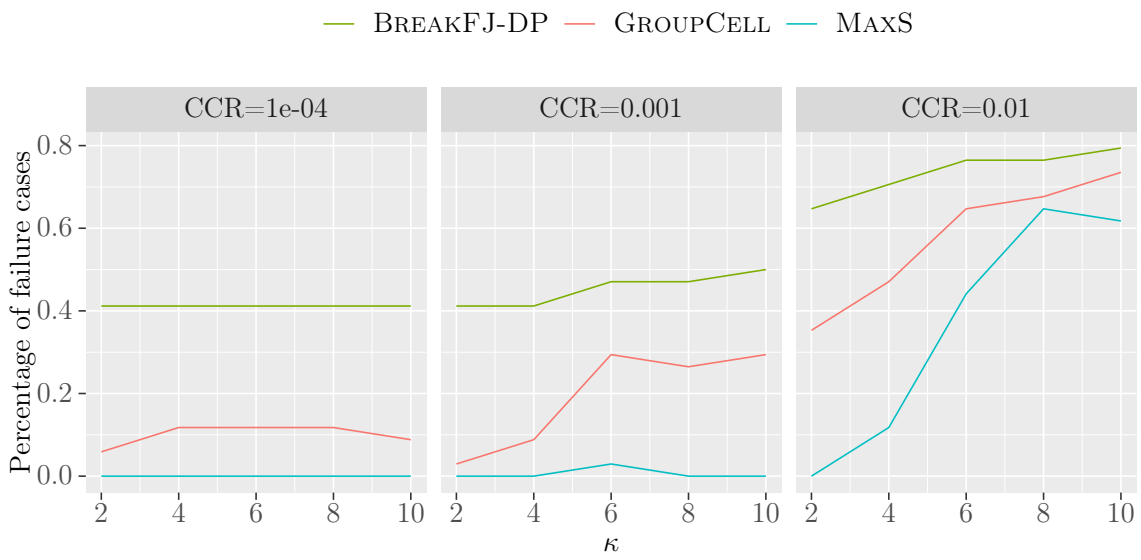


(b) Linear chains. 2 blocks, each with 4 cores provided. BreakFJ-DP, DP and GroupCell have the same number of failure cases as MaxS when CCR=10^{-2}, hence they are covered by MaxS. BreakFJ-DP is covered by DP when CCR=10^{-3}.

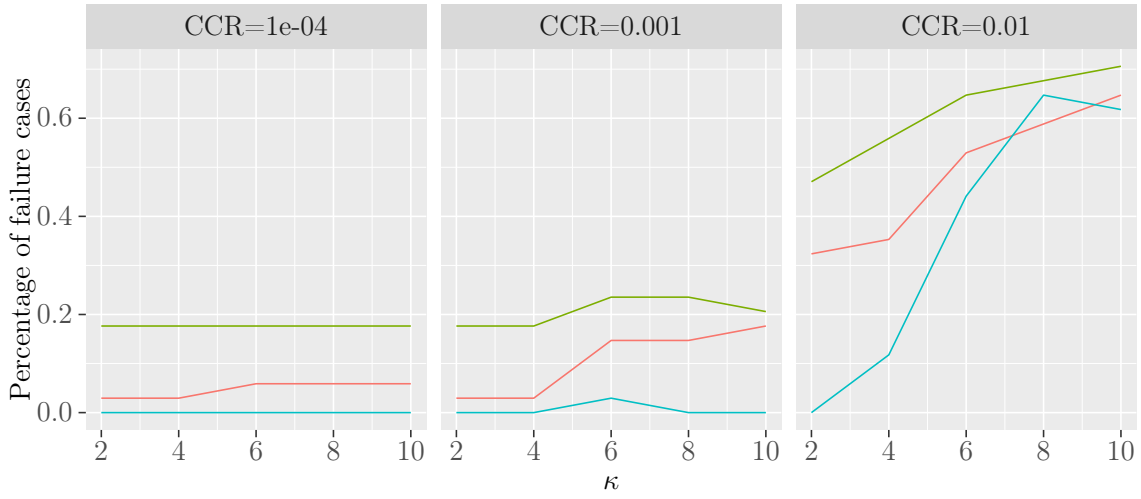


(c) Linear chains. 2 blocks, each with 8 cores provided. BreakFJ-DP, DP have the same number of failure cases as MaxS when CCR=10^{-4}, hence they are covered by MaxS. BreakFJ-DP and DP are covered by GroupCell when CCR=10^{-3}.

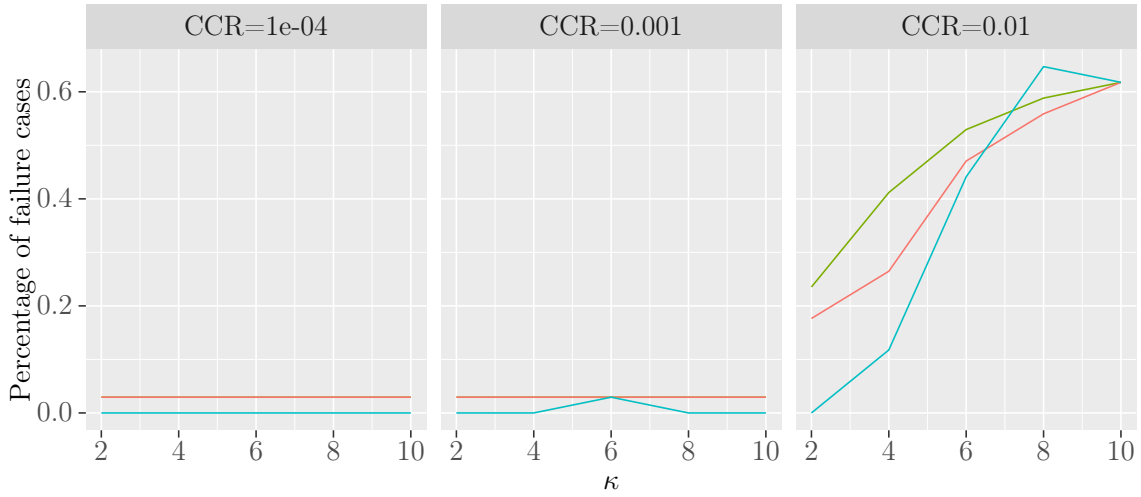
Figure 5.7 – Percentage of failure cases on linear chains as a function of κ.



(a) General SPGs. 4 blocks, each with 32 cores provided.



(b) General SPGs. 4 blocks, each with 64 cores provided.



(c) General SPGs. 4 blocks, each with 128 cores provided. BreakFJ-DP has the same number of failure cases as GroupCell when CCR=10^{-4} and 10^{-3}.

Figure 5.8 – Percentage of failure cases on general SPGs as a function of κ.

Chapter 6

Conclusions

In this thesis, we studied task mapping and load balancing problems in scheduling task graphs onto modern computing platforms, ranging from system on chips to distributed clusters. The history of using task graphs for parallel programming dates back to 1990s [73]. Relying on the explosion of computation demands and data dependencies of tasks by task graphs, sophisticated scheduling algorithms are proposed to fully exploit parallelism, task-based scheduling hence becomes prevalent in HPC today [84]. How to take fully advantage of multiprocessor systems to reduce the makespan and memory consumption of a tree of tasks is studied in Chapter 2. We then revisited in Chapter 3 a classical task-resource allocation problem, and proposed a novel dynamic scheduling algorithm that achieves better balance between data-locality and load-balancing. Mobile devices that are equipped with MPSoCs are more and more common now, and streaming applications running on these platforms demand intense computing power and real time performance. But they are often battery-operated and size-limited, so that low power (heat) and energy consumption have to be considered as well. We hence studied the scheduling of SPGs under a tight energy budget, high throughput and reliability constraints in Chapter 4 and Chapter 5. More detailed conclusions are listed in following.

Chapter 2: Partitioning tree-shaped task graphs for distributed platforms with limited memory. We studied a tree partitioning problem in Chapter 2. Computing demands and data dependencies of tasks are exhibited explicitly through this tree-shaped task graph. Processing the tree on one processor may exceeds local memory capacity because of the huge size of data. Partitioning the tree into many connected subtrees, and then processing each subtree on a processor equipped with a local memory is a feasible way to reduce the execution time and memory requirement. We formalized this problem and proved that it is NP-complete. Several efficient heuristics are proposed to tackle this problem in a reasonable time. We proposed a three-steps approach: 1) heuristics at the first step are focusing on partitioning the tree so as to reduce makespan; 2) at the second step, heuristics further partition subtrees that do not fit into local memory into smaller ones; 3) at the final step, some heuristics are proposed to merge smaller subtrees to form a feasible solution if more subtrees are generated than the number of available processors; or to further decompose subtrees to improve the makespan if there are idle processors left. Simulations on real task trees demonstrated that our heuristics can achieve up to 4 times improvement in makespan compared to a reference heuristic.

Chapter 3: Improving mappings for sparse direct solvers: a trade-off between data locality and load balancing. One of the pre-processing stages of sparse direct solvers consists of mapping computational resources (processors) to nodes of elimination trees. We revisit this problem in Chapter 3. The objective is to minimize the factorization time by exhibiting good data locality and load balancing. The proportional mapping technique is a widely used approach to solve this resource-allocation problem. It achieves

good data locality by assigning the same processors to large parts of the elimination tree. However, it may limit load balancing in some cases. We proposed a dynamic mapping algorithm based on proportional mapping. This new approach relaxes the data locality criterion to improve load balancing. Extensive experiments on a real world sparse matrix direct solver PaStiX demonstrated that our algorithm enables a better static scheduling of the numerical factorization while keeping good data locality.

Chapter 4: Reliability-aware energy optimization for throughput-constrained applications on MPSoC. Scheduling streaming applications onto a homogeneous MPSoC is considered in Chapter 4. The streaming application is modeled as a linear chain. Two metrics are considered in this work: 1) period bound, where data arrives continuously at this rate, the whole system has to execute datasets in this bound to avoid congestion. 2) the probability that the actual period exceeds the target period because of soft-errors in processors. The second metric corresponds to a real time constraint. To cope with radiation-induced soft-errors, we proposed a new model that allows us to change the frequency of the cores for different tasks and to duplicate some tasks. We proved that minimizing the energy consumption is easy without performance and reliability constraints, but that the problem becomes NP-complete when adding these constraints and when considering a discrete set of possible speeds. Several heuristics are proposed to solve this problem, in both continuous speed and more realistic discrete speed models. A heuristic named BestTrade can meet the two bounds without high energy consumption.

Chapter 5: Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms. Chapter 5 is a following work of Chapter 4. In this work, streaming applications are modeled as SPGs, which are more general than linear chains. The reliability bound is relaxed, very few errors that occur at the maximum speed is acceptable. To meet the tight energy budget while keeping the high reliability, triplication and majority voting is adopted in our model. We partition and map SPGs onto a computing platform with a hierarchical communication structure. Several cores are integrated by a low latency interconnect fabric in a block, and then blocks are connected by a slower interconnection. The goal is to minimize the energy cost while meeting the period bound and reliability bound. This problem has been proven NP-complete. Some practical solutions building upon a dynamic programming algorithm perspectives, which returns the optimal contiguous mapping for a linear chain are proposed.

Based on our work mentioned before, we propose some further directions for future work, both for short-term and long-term.

Short-term perspectives

Guarantee performance of heuristics. A further work for makespan minimization tree partition under memory constraint (Chapter 2) is to guarantee some of the heuristics performance and prove some approximation factors. The goal would be to show how bad the worst case can be compared to the optimal. For instance, for the problem of partitioning trees such that subtrees fit in local memory, a guarantee of how many subtrees the heuristic will produce is very useful to figure out if there is a viable partition scheme with a given number of processors. If the heuristic returns more subtrees than processors, say it returns s subtrees, and we have p processors, according to the ratio β between result from heuristics and from optimal, the number of subtrees requested by an optimal algorithm is then $(\frac{s}{1+\beta})$, we can infer that there is no viable partition if $(\frac{s}{1+\beta}) > p$. At the same time, it gives a standard to assess the absolute performance of heuristics instead of only comparing their performance to each other.

In Chapter 4, for each constraint, we proposed some heuristics. BestTrade is designed for meeting the probability bound, but the simulation results show that it also meets the

period bound. Is this a coincidence? Or is it because this strategy is also reasonable for achieving a high throughput such that the period bound could always be meet? We also would like to propose some heuristics that consider both constraints.

Validate our method on distributed settings. At the time of writing this thesis, PaStiX has only recently been extended to work on distributed settings, in order to assess the performance of Steal (or StealLocal) on the numerical factorization in distributed environments, hence performing further experiments on distributed platforms is natural when PaStiX will have stable performance on distributed settings.

Dynamic programming for homogeneous architecture. The dynamic programming approach proposed for scheduling applications that have the form of linear chains is optimal for monotonic mapping (see details in Chapter 5). To apply dynamic programming method on general SPGs, we have to consider the relative position of cores assigned to current task to cores mapped to its neighbor tasks (i.e., is it on the same core or same block, or neither) and if any of them are triplicated nor not. It makes the formula of combining optimal solutions of sub-problems very complicated. Because of its complexity, we gave up on applying a dynamic programming approach on general SPGs. If the computing platform is formed through a homogeneous communication system (e.g., multi/many cores chips), then we only need to consider if two neighbor tasks are mapped onto the same core or not. Then with a dedicated partitioning rule, for example, a fork node should be mapped onto the same core as its predecessor, the dynamic programming may be applicable for general SPGs. In conclusion, we could explore a dynamic programming approach on general SPGs for a homogeneous computing platform.

Long-term perspectives

Open problems in tree partitioning. Graph partitioning has been known as a complex problem, extensive endeavors have been devoted to propose some efficient solutions. But still lots of open questions are waiting for answers. While we have focused so far on partitioning trees to optimize the execution time under memory constraints in Chapter 2, some other optimization problems could be defined in this context:

- MinMakespan: minimize the makespan, i.e., the total execution time, given a set of p processors (i.e., we must have a number of subtrees $k \leq p$);
- MinNbSubtrees: minimize the number of subtrees k, without considering the makespan.

We have already proved that MinMakespan is NP-complete. For both problems listed above, there is still a lot of work to do. The main research objectives can be summarized in the following questions:

1. Without the memory constraint, i.e., when the whole tree fits in the memory M, what is the complexity of MinMakespan? Is it still an NP-complete problem?
2. Is MinNbSubtrees a hard problem? How can we find the minimum number of subtrees required so that each subtree fits in memory? This amounts to finding the minimum number of processors required to execute the tree.

Taking advantage of heterogeneous MPSoC. Processing elements are homogeneous so far in our model, they all have the same characteristics. It is anticipated that heterogenous cores and accelerators are one of the key engines to achieve performance scaling and energy efficiency [18]. Some industrial MPSoC products like Zynq UltraScale+ [1], provide efficient power management through many power islands. Multiple power domains exist on the chip, each with its own power management features, so a flexible block-level power

management can be achieved. APUs (Application Processor Units) and on chip GPUs (Graphics Processing Units) are in full-power domain, whereas RPUs (Real-time Processor Units) are in low-power domain. PL (Programmable Logic) has its own specific power domain. RPUs can be totally shut down, then the static power of this unused block is eliminated. Selecting the right processing elements for different tasks plays a key role on power optimization. Typically, APUs are favourable for data processing. RPUs can process some real-time events with potentially lower latency and lower static power.

Some streaming applications consist of different types of tasks that exhibit different types of operations, memory bandwidth and access patterns [90]. We expect more and more task graphs with such details of task's type in the near future. It would be helpful in assigning them to the most suitable processing elements to achieve a high performance per watt. Given task graphs with details in task's type and a highly heterogeneous MPSoC, an interesting direction could be energy minimization under performance constraint through the selection of the best processing elements for each task.

Bibliography

[1] Managing power and performance with the Zynq UltraScale+ MPSoC, October 2016.

[2] Emmanuel Agullo, George Bosilca, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Exploiting a parametrized task graph model for the parallelization of a sparse direct multifrontal solver. In *Euro-Par 2016: Parallel Processing Workshops*, pages 175–186. Springer International Publishing, 2017.

[3] S. Albers and M. Hellwig. Online makespan minimization with parallel schedules. *Algorithmica*, 78:492–520, 2017.

[4] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J. Y. L'Excellent, and B. Uçar. MUMPS. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1232–1238. Springer, 2011.

[5] P. R. Amestoy, I. S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2):501 – 520, 2000.

[6] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[7] I. Assayad, A. Girault, and H. Kalla. Tradeoff exploration between reliability, power consumption, and execution time for embedded systems. *International Journal on Software Tools for Technology Transfer*, 15:229–245, 2013.

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

[9] Guillaume Aupy and Anne Benoit. Approximation algorithms for energy, reliability, and makespan optimization problems. *Parallel Process. Lett.*, 26(1):1650001:1–1650001:23, 2016.

[10] J. Ax, G. Sievers, J. Daberkow, M. Flasskamp, M. Vohrmann, T. Jungeblut, W. Kelly, M. Porrmann, and U. Rückert. CoreVA-MPSoC: A many-core architecture with tightly coupled shared and local data memories. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1030–1043, 2018.

[11] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proc. of Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2003.

[12] Olivier Beaumont and Abdou Guermouche. Task scheduling for parallel multifrontal methods. In *European Conference on Parallel Processing*, pages 758–766. Springer, 2007.

[13] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *J. Parallel and Distributed Computing*, 68(6):790–808, 2008.

[14] Shishir Bharathi and Ann Chervenak. Scheduling data-intensive workflows on storage constrained resources. In *Proc. of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*. ACM, 2009.

[15] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[16] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[17] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, and al. KiloCore: A fine-grained 1,000-processor array for task-parallel applications. *IEEE Micro*, 37:63–69, 2017.

[18] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[19] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[20] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *CoRR*, abs/0709.1272, 2007.

[21] Vincent Cavé, Romain Clédat, Paul Griffin, Ankit More, Bala Seshasayee, Shekhar Borkar, Sanjay Chatterjee, Dave Dunning, and Joshua Fryman. Traleika glacier: A hardware-software co-designed approach to exascale computing. *Parallel Computing*, 64:33 – 49, 2017.

[22] CERN. https://home.cern/science/computing/processing-what-record.

[23] E. Chan, F. G. Van Zee, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Satisfying your dependencies with SuperMatrix. In *2007 IEEE International Conference on Cluster Computing*, pages 91–99, Sept 2007.

[24] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 123–132, New York, NY, USA, 2008. Association for Computing Machinery.

[25] G. Chen, K. Huang, and A. Knoll. Energy Optimization for Real-time Multiprocessor System-on-chip with Optimal DVFS and DPM Combination. *ACM Trans. on Embedded Computing Systems*, 13:111:1–111:21, 2014.

[26] CMS Collaboration. CMS data processing workflows during an extended cosmic ray run. *Journal of Instrumentation*, 5(03):T03006–T03006, mar 2010.

[27] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele. Combined DVFS and Mapping Exploration for Lifetime and Soft-error Susceptibility Improvement in MPSoCs. In *Proc. of Design, Automation & Test in Europe (DATE)*, pages 61:1–61:6, 2014.

[28] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[29] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, 2006.

[30] Jack Deslippe, Abdelilah Essiari, Simon J. Patton, Taghrid Samak, Craig E. Tull, Alexander Hexemer, Dinesh Kumar, Dilworth Parkinson, and Polite Stewart. Workflow management for real-time analysis of lightsource experiments. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, WORKS '14, pages 31–40. IEEE Press, 2014.

[31] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555 – 1581, 2000.

[32] A. Dolgert, L. Gibbons, C. D. Jones, V. Kuznetsov, M. Riedewald, D. Riley, G. J. Sharp, and P. Wittich. Provenance in high-energy physics workflows. *Computing in Science Engineering*, 10(3):22–29, May 2008.

[33] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, Sept 1973.

[34] W.E. Donath and A.J. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[35] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, pages 280–288, 2007.

[36] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.*, 9(3):302–325, September 1983.

[37] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.

[38] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Parallel scheduling of task trees with limited memory. *ACM Transactions on Parallel Computing*, 2(2):13, 2015.

[39] Andreas Emil Feldmann. Fast balanced partitioning of grid graphs is hard. *CoRR*, abs/1111.6745, 2011.

[40] Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, Feb 2015.

[41] G. P. Fettweis. The tactile internet: Applications and challenges. *IEEE Vehicular Technology Magazine*, 9:64–70, 2014.

[42] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, June 1982.

[43] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co, London (UK), 1979.

[44] Alan George, Joseph WH Liu, and Esmond Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, 1989.

[45] M. A. Haque, H. Aydin, and D. Zhu. On reliability management of energy-aware real-time systems through task replication. *IEEE Trans. on Parallel and Distributed Systems*, 28(3):813–825, 2017.

[46] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Rev.*, 33(3):420–460, August 1991.

[47] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[48] J. Hu and R. Marculescu. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, 2005.

[49] K. Huang, W. Haid, L. Bacivarov, M. Keller, and L. Thiele. Embedding Formal Performance Analysis into the Design Cycle of MPSoCs for Real-time Streaming Applications. *ACM Trans. on Embedded Computing Systems*, 11:8:1–8:23, 2012.

[50] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS 2011, 25th IEEE International Symposium on Parallel and Distributed Processing*, pages 556–567, Anchorage, Alaska, USA, 2011. IEEE Computer Society.

[51] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine A. Yelick. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. *CoRR*, 2016.

[52] W. Jeong, P. T. Fletcher, R. Tao, and R. Whitaker. Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware hamilton-jacobi solver. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1480–1487, 2007.

[53] W. Kelly, M. Flaßkamp, G. Sievers, J. Ax, J. Chen, C. Klarhorst, C. Ragg, T. Jungeblut, and A. Sorensen. A communication model and partitioning algorithm for streaming applications for an embedded MPSoC. In *2014 International Symposium on System-on-Chip (SoC)*, pages 1–6, Oct 2014.

[54] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[55] Kyungjoo Kim, H. Carter Edwards, and Sivasankaran Rajamanickam. Tacho: Memory-scalable task parallel sparse cholesky factorization. In *IPDPS Workshops*, pages 550–559. IEEE Computer Society, 2018.

[56] Kyungjoo Kim and Victor Eijkhout. A parallel sparse direct solver via hierarchical DAG scheduling. *ACM Trans. Math. Softw.*, 41(1):3:1–3:27, October 2014.

[57] Chi-Chung Lam, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, 2011.

[58] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. *50 Years of Integer Programming 1958-2008*, pages 105–132, 2010.

[59] K. Li, X. Tang, and K. Li. Energy-efficient stochastic task scheduling on heterogeneous computing systems. *IEEE Trans. on Parallel and Distributed Systems*, 25(11):2867–2876, 2014.

[60] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

[61] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12(3):249–264, 1986.

[62] Joseph W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8(3), 1987.

[63] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[64] Xiao Liu, Yun Yang, Dong Yuan, and Jinjun Chen. Do we need to handle every temporal violation in scientific workflow systems? *ACM Trans. Softw. Eng. Methodol.*, 23(1), February 2014.

[65] László Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2:1–46, 1993.

[66] Andre Luckow, George Chantzialexiou, and Shantenu Jha. Pilot-streaming: A stream processing framework for high-performance computing, 2018.

[67] P. Marwedel, J. Teich, G. Kouveli, L. Bacivarov, L. Thiele, and et al. Mapping of Applications to MPSoCs. In *Proc. of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 109–118, 2011.

[68] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 538–547, Washington, DC, USA, 1991. IEEE Computer Society.

[69] G. Onnebrink, F. Walbroel, J. Klimt, R. Leupers, G. Ascheid, L. G. Murillo, S. Schürmans, X. Chen, and Y. Harn. DVFS-enabled power-performance trade-off in MPSoC SW application mapping. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 196–202, July 2017.

[70] Alex Pothen and Chunguang Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[71] GN Srinivasa Prasanna and Bruce R. Musicus. Generalized multiprocessor scheduling and applications to matrix computations. *IEEE TPDS*, 7(6):650–664, 1996.

[72] Arun Ramakrishnan, Gurmeet Singh, Henan Zhao, Ewa Deelman, Rizos Sakellariou, Karan Vahi, Kent Blackburn, David Meyers, and Michael Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *CCGrid'07*, pages 401–409, 2007.

[73] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[74] E. Rothburg and A. Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 503–512, 1993.

[75] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. Comput.*, 38(1):62–81, January 1989.

[76] P. Sao, X. S. Li, and R. Vuduc. A communication-avoiding 3D LU factorization algorithm for sparse matrices. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 908–919, May 2018.

[77] S. Schamberger. On partitioning FEM graphs using diffusion. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 277–286, April 2004.

[78] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the intel Skylake-SP processor and their impact on performance. *CoRR*, abs/1905.12468, 2019.

[79] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135 – 148, 1991.

[80] J. Spasic, D. Liu, and T. Stefanov. Energy-efficient mapping of real-time applications on heterogeneous MPSoCs using task replication. In *Proc. of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2016.

[81] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[82] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer. Top500 lists, https://www.top500.org/lists/.

[83] Q. Tang, S. Wu, J. Shi, and J. Wei. Optimization of duplication-based schedules on network-on-chip based multi-processor system-on-chips. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):826–837, March 2017.

[84] Samuel Thibault. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Habilitation à diriger des recherches, Université de Bordeaux, December 2018.

[85] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, MIT, Cambridge, MA, USA, 2009.

[86] D. Truong, W. Cheng, T. Mohsenin, and et al. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *Proc. of IEEE Symposium on VLSI Circuits*, pages 22–23, 2008.

[87] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. J. Garzarán. Mapping streaming applications on commodity multi-CPU and GPU on-chip processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1099–1115, April 2016.

[88] S. Wang, K. Li, J. Mei, G. Xiao, and K. Li. A reliability-aware task scheduling algorithm based on replication on heterogeneous computing systems. *Journal of Grid Computing*, 15(1):23–39, Mar 2017.

[89] Xiaofeng Wang, Chee Shin Yeo, Rajkumar Buyya, and Jinshu Su. Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm. *Future Gener. Comput. Syst.*, 27(8):1124–1134, October 2011.

[90] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[91] Fang Xu and Klaus Mueller. Real-time 3D computed tomographic reconstruction using commodity graphics hardware. *Phys Med Biol*, 52(12):3405–3419, Jun 2007.

[92] Hongzhi Xu, Renfa Li, Chen Pan, and Keqin Li. Minimizing energy consumption with reliability goal on heterogeneous embedded systems. *Journal of Parallel and Distributed Computing*, 127:44 – 57, 2019.

[93] Asim YarKhan. *Dynamic task execution on shared and distributed memory architectures*. PhD thesis, University of Tenessee, 2012.

[94] L. Zhang, K. Li, C. Li, and K. Li. Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems. *Information Sciences*, 379:241–256, 2017.

[95] S. Zhang, J. Wu, and S. Lu. Distributed workload dissemination for makespan minimization in disruption tolerant networks. *IEEE Transactions on Mobile Computing*, 15:1661–1673, 2016.

[96] B. Zhao, H. Aydin, and D. Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Proc. of Design Automation Conference (DAC)*, pages 381–386, 2011.

[97] W. Zheng and R. Sakellariou. Stochastic DAG scheduling using a Monte Carlo approach. *Journal of Parallel and Distributed Computing*, 73(12):1673 – 1689, 2013.

[98] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. *ACM Trans. on Embedded Computing Systems*, 10:26:1–26:27, 2011.

[99] Dakai Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '04, pages 35–40, USA, 2004. IEEE Computer Society.

List of publications

Articles in International Refereed Journals

[J1] Changjiang Gou, Anne Benoit, and Loris Marchal. Partitioning tree-shaped task graphs for distributed platforms with limited memory. *IEEE Trans. Parallel Distrib. Syst.*, 31(7):1533–1544, 2020.

Articles in International Refereed Conferences

[C1] Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, and Tongquan Wei. Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms. In *32nd IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2020, Porto, Portugal, September 8-11, 2020*. SBAC-PAD, IEEExplore, 2020.

[C2] Changjiang Gou, Ali Al Zoobi, Anne Benoit, Mathieu Faverge, Loris Marchal, Grégoire Pichon, and Pierre Ramet. Improving mapping for sparse direct solvers - A trade-off between data locality and load balancing. In Maciej Malawski and Krzysztof Rzadca, editors, *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings*, volume 12247 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2020.

[C3] Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, and Tongquan Wei. Reliability-aware energy optimization for throughput-constrained applications on MP-SoC. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*, pages 577–586. IEEE, 2018.

[C4] Changjiang Gou, Anne Benoit, and Loris Marchal. Memory-aware tree partitioning on homogeneous platforms. In Ivan Merelli, Pietro Liò, and Igor V. Kotenko, editors, *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, pages 321–324. IEEE Computer Society, 2018.

Research Reports

[R1] Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, and Tongquan Wei. Reliable and energy-aware mapping of streaming series-parallel applications onto hierarchical platforms. Research Report RR-9346, INRIA, June 2020.

[R2] Changjiang Gou, Ali Al Zoobi, Anne Benoit, Mathieu Faverge, Loris Marchal, Grégoire Pichon, and Pierre Ramet. Improving mapping for sparse direct solvers: A trade-off between data locality and load balancing. Research Report RR-9328, Inria Rhône-Alpes, February 2020.

[R3] Anne Benoit, Changjiang Gou, and Loris Marchal. Partitioning tree-shaped task graphs for distributed platforms with limited memory. Research Report RR-9115, Inria Grenoble Rhône-Alpes, March 2019.

[R4] Changjiang Gou, Anne Benoit, Mingsong Chen, Loris Marchal, and Tongquan Wei. Reliability-aware energy optimization for throughput-constrained applications on MP-SoC. Research Report RR-9168, INRIA;ECNU;Georgia Tech., April 2018.