Energy-aware scheduling : complexity and algorithms

Paul Renaud-Goud

▶ To cite this version:

Paul Renaud-Goud. Energy-aware scheduling : complexity and algorithms. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2012. English. NNT : 2012ENSL0727 . tel-00744247

HAL Id: tel-00744247
https://theses.hal.science/tel-00744247v1

Submitted on 22 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° attribué par la bibliothèque : 2012ENSL0727

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -

Laboratoire de l'Informatique du Parallélisme - UMR5668 - LIP

THÈSE

en vue d'obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon - Université de Lyon

Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 5 juillet 2012 par

Paul RENAUD-GOUD

Energy-aware scheduling: complexity and algorithms

Directeur de thèse :	Anne	BENOIT	
Co-encadrant de thèse :	Yves	ROBERT	
Après avis de :	Olivier	BEAUMONT	*Rapporteur*
	Padma	RAGHAVAN	*Rapportrice*
Devant la commission d'examen formée de :			
	Olivier	BEAUMONT	*Rapporteur*
	Anne	BENOIT	*Membre*
	Jean-Marc	PIERSON	*Examinateur*
	Padma	RAGHAVAN	*Rapportrice*
	Yves	ROBERT	*Membre*
	Denis	TRYSTRAM	*Examinateur*

Remerciements

Un jour, Benoît, un ami, m'a expliqué que les remerciements étaient la partie la plus importante d'un manuscrit de thèse. Croyez bien que je ne le remercie pas. A l'inverse, je remercie Laura de m'avoir fait découvrir l'incroyable similitude entre notre formule d'énergie et $E = mc^2$. C'est fou.

Je me dois évidemment de remercier Olivier Beaumont et Padma Raghavan pour leur travail. Je les remercie également, ainsi que Jean-Marc Pierson et Denis Trystram, pour leur venue à ma soutenance de thèse, et leurs questions pertinentes. Et merci à Anne et Yves pour ces trois années de science (et autres) à leurs côtés.

Je remercie chaleureusement toutes les personnes qui ont partagé mon bureau, leur bureau, ou le balcon. Et pour terminer, je ne peux que remercier le lecteur perdu qui s'est donné la peine de lire les remerciements, et peut-être même un chapitre.

Acknowledgements

I would like to thank Olivier Beaumont and Padma Raghavan, my two reviewers, for their work, my committee chairman, Denis Trystram, and Jean-Marc Pierson for joining my thesis committee. I thank all of them for attending my defense, and especially Padma Raghavan who had a long way to go.

Contents

Introduction **i**

1 On the performance of greedy algorithms **1**
1.1 Introduction . . . 1
1.2 Related work . . . 4
1.3 Main contributions . . . 5
1.4 Proof of the main theorems . . . 5
1.5 The approximation factor as a function of p . . . 14
1.6 Conclusion . . . 15

2 Mapping concurrent streaming applications **17**
2.1 Introduction . . . 17
2.2 Related work . . . 19
2.3 Motivating example . . . 20
2.3.1 Interval mappings . . . 20
2.3.2 General mappings . . . 21
2.4 Framework . . . 22
2.4.1 Applicative framework . . . 22
2.4.2 Target platform . . . 22
2.4.3 Mapping strategies and scheduling . . . 23
2.4.4 Performance optimization criteria . . . 23
2.4.5 Energy model . . . 26
2.5 Complexity results with the PATH model . . . 26
2.5.1 Period minimization . . . 27
2.5.2 Latency minimization . . . 31
2.5.3 Period/latency minimization . . . 32
2.5.4 Period/energy minimization . . . 34
2.5.5 Period/latency/energy minimization . . . 35
2.5.6 Summary of complexity results for the PATH model . . . 39
2.6 Complexity results with the WAVEFRONT model . . . 40
2.6.1 Period minimization . . . 40
2.6.2 Period/latency minimization . . . 41
2.6.3 Period/latency/energy minimization . . . 41
2.7 Simulations with the WAVEFRONT model . . . 42
2.7.1 Integer linear program . . . 43
2.7.2 Heuristics . . . 45
2.7.3 Simulation results . . . 49

2.8 Conclusion . . . 53

3 Replica placement and update strategies in tree networks 55
3.1 Introduction . . . 55
3.2 Framework . . . 56
3.2.1 Replica servers . . . 56
3.2.2 With power consumption modes . . . 57
3.2.3 Objective functions . . . 58
3.2.4 Summary of results . . . 58
3.3 Complexity results: update strategies . . . 59
3.3.1 Running example . . . 59
3.3.2 Dynamic programming algorithm . . . 60
3.3.3 Execution time of the algorithm . . . 64
3.4 Complexity results with power . . . 64
3.4.1 Running example . . . 64
3.4.2 NP-completeness of MINPOWER . . . 65
3.4.3 A pseudo-polynomial algorithm for MINPOWER-BOUNDEDCOST . . . 68
3.5 Simulations . . . 68
3.5.1 Impact of pre-existing servers . . . 68
3.5.2 With power consumption . . . 70
3.5.3 Running time of the algorithms . . . 72
3.6 Conclusion . . . 72

4 Mapping series-parallel workflows onto CMPs 75
4.1 Introduction . . . 75
4.2 Related work . . . 76
4.3 Framework . . . 78
4.3.1 Applicative framework . . . 78
4.3.2 Platform . . . 79
4.3.3 Mapping strategies . . . 80
4.3.4 Period . . . 81
4.3.5 Energy model . . . 81
4.4 Complexity results . . . 82
4.4.1 Uni-directional uni-line CMP . . . 82
4.4.2 Bi-directional uni-line CMP . . . 84
4.4.3 Square CMP . . . 85
4.4.4 Integer linear program . . . 88
4.5 Heuristics . . . 90
4.5.1 Random heuristic . . . 90
4.5.2 Greedy heuristic . . . 91
4.5.3 2D dynamic programming algorithm . . . 91
4.5.4 1D heuristics . . . 93
4.6 Simulation results . . . 93
4.6.1 Simulation setting . . . 93
4.6.2 Simulation results . . . 94
4.7 Conclusion . . . 103

5 Manhattan routing on CMPs **105**
5.1 Introduction . . . 105
5.2 Related work . . . 106
5.3 Framework . . . 107
5.3.1 Platform and power consumption model . . . 107
5.3.2 Communications . . . 107
5.3.3 Routing rules . . . 108
5.3.4 Problem definition . . . 109
5.3.5 Comparison of routing rules . . . 109
5.4 Theoretical results . . . 110
5.4.1 Manhattan vs XY . . . 110
5.4.2 NP-completeness . . . 116
5.5 Heuristics . . . 117
5.5.1 Simple greedy (**SG**) . . . 117
5.5.2 Improved greedy (**IG**) . . . 117
5.5.3 Two-bend (**TB**) . . . 118
5.5.4 XY improver (**XYI**) . . . 118
5.5.5 Path remover (**PR**) . . . 118
5.6 Simulations . . . 119
5.6.1 Sensitivity to the number of communications . . . 119
5.6.2 Sensitivity to the size of communications . . . 120
5.6.3 Sensitivity to the average length of communications . . . 121
5.6.4 Summary of simulations . . . 122
5.7 Conclusion . . . 123

6 Assessment of bi-criteria heuristics for general DAGs **125**
6.1 Introduction . . . 125
6.2 Related work . . . 126
6.3 Framework . . . 126
6.3.1 DAG . . . 126
6.3.2 Platform . . . 127
6.3.3 Frequency scaling strategies . . . 127
6.3.4 Energy model . . . 127
6.4 Slack reclamation algorithms . . . 128
6.4.1 Mapping algorithm: **HEFT** . . . 128
6.4.2 **LPHM** . . . 128
6.4.3 **SRP** . . . 128
6.4.4 **LEneS** . . . 130
6.4.5 **Opt** . . . 133
6.5 Simulations . . . 135
6.5.1 Fatness and Communication-to-computation ratio . . . 136
6.5.2 Number of frequencies . . . 136
6.5.3 Graph size . . . 137
6.5.4 Conclusion of the simulations . . . 140
6.6 Conclusion . . . 140

Conclusion **143**

Bibliography **149**

Publications **159**

Introduction

Computer designers have always tried to improve the capacity of their products, and as of now, they have always succeeded. Computers become smaller and smaller, they can store more and more data, they compute faster and faster, etc. Pushed by an extension of the Moore's law, constructors have notably guaranteed that the computational power of the processors would double every 18 months.

Unfortunately, this computational power is deeply related to the chip power supply. The miniaturization of the chip, as well as the increase of the clock frequency, have led to a consequent rise of the dissipated power and the consumed energy. This recent evolution has rendered the energy issue crucial, both locally — in the small chip — and globally — in the wide world.

On the microscopic scale, the power density is beginning to be worrying, since it becomes close to the power of a nuclear reactor [116]. One of the consequences is that the temperature reaches an excessive level, which in particular weakens drastically the reliability of the chip.

On the world scale, when Mills invoked in 1999 the idea that the popularization of the computers, as well as their increasing need of electricity, would end up in a real energy issue [86], he encountered virulent criticisms, in particular by other researchers. One should point out today that he was not wrong.

The energy problem affects every computer user, from the company that uses data centers, to the lambda-user who owns his laptop at home, and including enterprises and research institutes that build supercomputers. Google revealed recently its power consumption during the year 2011: 260 million Watts, that is the whole output of a power plant, or about a quarter of the output of a nuclear plant. More generally the Climate Group and the Global e-Sustainability Initiative released in 2008 a report on cloud computing [92], which studies among others the electricity consumption of worldwide data centers. In 2007 they consumed 330 billion Watts hour, which is higher than the electricty consumption of such a big country like France. The current estimation gives an energy consumption of around 1000 billion Watts hour during the year 2020, which would overpass Russia in the ranking of countries consuming the most energy for electricity end, given that Russia is the third country in the list.

Those numbers about energy consumption can be translated as bad impacters on the environment. Also, according to a report by Greenpeace [48], the data centers are expected to reject the equivalent of 533 million tonnes of $\mathrm{CO_2}$ equivalent because of their electricity consumption, which comes mostly from dirty energies. If they were a country, they would end up in the top ten of the $\mathrm{CO_2}$ producers.

The computer scientists who work in high-performance computing expect to run an Exascale machine, able to compute 10^{18} floating operations per second, in the 2020's. The first previsions show that if no effort is done concerning the energy consumption, it will be impossible for such an Exascale machine to exist. In addition, as said in "The IESP road map" [41], a reasonable power dissipation is estimated to be around $100\,MW$. Based on the cheapest electricity cost – $1\,W$ during $1\,year$ at \$1, this means that every single percent of power saved will save 1 million dollar each year.

One can wonder if the energy issue is relevant for a basic user that turns on its small personal machine at home after work. A study (see [61]) shows that a personal computer is consuming $500\,kWh$ per year in average. Given that more than 1 billion personal computers populate the earth, their electricity

consumption reaches 500 trillion Watts hour per year. And this is getting worse since 4 billions personal computers are expected in 2020.

The energy that feeds a computer is shared between different parts, such like disk drive, motherboard, fans, etc. But the most energy-hungry component is the processor: even idle, it dissipates at least half of the total power. That is why we only consider this energy in this thesis. In single core processors, this energy is the energy consumed by the Central Processing Unit. The most commonly used model about the power dissipated by a CPU is given in the following paragraph. In multi-core processors, a part of the power is dissipated in routers that ensure the communication between cores; the model that we use makes the power dissipated by CPUs (for the computations) symmetrical to the power dissipated by routers (for the communications).

The consumed power in a CPU is divided into a static part P_{stat} and a dynamic part P_{dyn}. The static part is the cost for a processor to be on, whereas the dynamic part is an additional cost for the computations, according to the speed at which the processor is running. More precisely, we have: $P_{stat} = VI_{leak}$ and $P_{dyn} = aCV^2f$, where:

- a is the switching activity: the chip is composed of transistors, that switches during a computation;
- C is the physical capacitance;
- V is the supply voltage;
- f is the clock frequency, or the number of clock tops per second;
- I_{leak} is the leakage current, due to the nature of the transistors.

A lot of research has been done with the aim to reduce power consumption through hardware modifications, and tries to act on all those factors. Transistors have been improved, in order to reduce a and C. The temperature plays an important role, through the current leakage. When the temperature is rising, the current leakage is increasing, thus the dissipated power is higher, which leads to an elevation of the temperature. This vicious circle must be avoided, thanks to effective cooling systems. But the saved power in the processor must be greater than the power needed to make the cooling system work. That is why surprising cooling architectures appear sometimes, like from Google, which wants to install a data center on a boat to refresh it at least cost.

The most important breakthrough is the advent of the **D**ynamic **V**oltage and **F**requency **S**caling (DVFS), which is enabled on almost all recent processors: the new generation is indeed able to run at different speeds and voltages, those parameters being set up by the user. The name of this technique is a little confusing: from an energy perspective, we can think that we just have to set the voltage to its minimum and the frequency to its maximum to obtain an excellent ratio performance versus dissipated power. This is not so easy, since only a set of voltages is reachable, and each voltage is associated with a set of possible frequencies. Of course, the upper the supply voltage, the higher the running frequency can be. This reality is often simplified through an idealized model. Actually, several models exist to describe the same reality. It is commonly assumed that the voltage is proportional to the frequency, implying a dynamic power in the cube of the frequency. Concerning the set of possible frequencies, it can be either continuous — the frequency stays anywhere between a minimum frequency and a maximum frequency — or discrete — we are given a set of possible frequencies.

Finally, a simple but efficient way to reduce the consumed power, while maintaining a given level of performance, is to divide the computational units into smaller ones. This idea is exploited in the **C**hip **M**ulti**P**rocessors (CMPs): two processing units at frequency $f/2$ dissipate a power $f^3/4$, whereas one bigger processing unit at frequency f dissipates a power f^3, for the same number of operations per second.

Equipped with the power dissipation model and the promising DVFS method, there just remains to convert the try: deriving algorithms that squeeze the most out of this potential energy saving. This technique should open algorithmists' mind in order to make the existing algorithms energy-aware and to design new algorithms that will be energy-oriented.

Because of the convexity of the power function as a function of the running frequency, if a processor cannot be turned off, it is always better to run its work as slow as possible. However, this solution would not be accepted by most people: performance degradation is not allowed in the computers world. In any domain, only a tiny minority would sacrifice the performance of their applications. Obviously, a supercomputer manager would deny it, since his primary goal is to compute always faster, but even a basic user would be furious if his machine was becoming slower.

Therefore we have to find smarter ways to decide for a scheduling policy. The frequencies must be balanced, again because of the convexity of the power function, so that the total execution time, or any performance-related criterion, is not strongly downgraded. Instead of fulfilling a tight constraint on the performance, we can indeed be given an interval of validity.

The reverse problem can also be stated: we dispose of a given amount of energy, and a performance-related criterion must be maximized. This problem takes place very naturally in useful applications, e.g. an unplugged laptop must run the maximum amount of work until its battery is empty, or the lifetime of an embedded device must be maximized.

Those two very general problems can be instanciated into numerous smaller ones. During this thesis we have been dealing with some of them, which we describe briefly below.

On the performance of greedy algorithms for power consumption minimization [B4]

We start with a simple and classical problem, accompanied by a simple and classical greedy algorithm. We are given a set of homogeneous processors and a set of independent jobs that are to be executed on the processors. We want to assign each job to a processor in the best way. This problem has been widely studied, when "best way" means "so that the totality of the jobs is finished as soon as possible", or in technical terms, "so that the makespan is minimum". In this case a natural heuristic consists in greedily assigning each job to the currently least loaded processor.

In this first chapter, we study a close problem, in which "best way" means "so that the processors consume as less energy as possible". We dispose of processors with continuous frequencies, and we must fulfill a constraint on the makespan, so that the energy consumed by a processor is equal to the cube of the sum of its computations size. Intuitively the previous algorithm seems reasonable, since it balances the computations. We exhibit lower bounds on the energy consumption of the algorithm, for online and offline versions: in this second version, jobs are sorted by non-increasing size. We show that those bounds are tight.

Mapping concurrent streaming applications [B7, B6, A1]

In this second chapter, a few complications occur. On the one hand, tasks are not any more independent: we have several applications, each of them being a linear chain of tasks – apart from the entry task and the exit task, each task receives data from its previous task and sends data to its successor task. On the other hand, we optimize now three criteria, namely energy, period and latency. The energy remains the energy consumed by the processors, and latency is a measure that is close to makespan and execution time. The applications we consider here are streaming applications: in other words, they never stop to

compute. A set of data enters the application through the entry task every period, and will be output by the exit task after some periods. This period is thus another performance-related criterion.

In order to understand precisely where the difficulty of the problem comes from, we define different way to map the tasks onto the processors, and establish the problem complexity under each mapping rule. The one-to-one mappings, in which a processor is assigned at most one task, are a restriction of the interval mappings, where a processor is assigned an interval of consecutive tasks. A general mapping does not suffer from any constraint. In addition to those different complexity levels of mappings, we lean on different heterogeneity degrees concerning the platform: homogeneous processors and homogeneous links between them, heterogeneous processors and homogeneous links, and fully heterogeneous platforms.

On the theoretical side, we perform an exhaustive complexity study of all mono-criterion, bi-criteria and tri-criteria problems, according to the mapping rules and the heterogeneity of the platform: we exhibit polynomial algorithms for simpler instances and NP-completeness proofs for more intricate ones. On the practical side, we design several heuristics that give a reasonable solution to the most general problem, and run simulations with randomly generated applications.

Replica placement and update strategies in tree networks [B5]

In this third chapter, the elements that need to be computed are no longer distributed along a linear chain but are located on the leaves of a tree. The leaves are called clients, and they send requests that need to be handled by a server upper in the tree. Such a problem finds applications in real-life: in Video on Demand services, users are requesting video files in the network. One server cannot serve all the user requests; therefore the initial file has to be replicated inside the tree, so that every user can receive its movie in acceptable time.

Let us describe more precisely the framework: the requests of the clients are counted per time unit, hence the requests a server handles are also per time unit. The objective function is to minimize the power consumed by the servers. Obviously, the more requests, the higher dissipated power. On top of this goal to save power, we add the notion of dynamicity. Sometimes the number of requests of a client might change, and the previous distribution of the servers may not be optimal. Despite that, moving a server or creating a new server might not be advantageous, since it implies turning on a new server and copying the file into its new location. Also we introduce a cost related to such server operations. Finally, the problem of this chapter is the following: we are given a tree, filled with clients, and provided with a set of pre-existing servers, and we want to minimize the dissipated power, without exceeding a given bound on the cost.

We study the complexity of the problem under different hypotheses, and we give notably an intricate pseudo-polynomial dynamic programming algorithm. To the best of our knowledge, we are the first to deal with power consumption and dynamicity on this problem, so we evaluate our algorithm by comparing with a classical algorithm that solves the simple replica placement problem.

Mapping series-parallel workflows onto chip multiprocessors [B3]

In this fourth chapter, we come back to streaming applications that have to be mapped onto a platform. We are given an application in the form of a task graph, which belongs to the series-parallel class of graphs. Since all the task graphs of streaming applications that we encountered were series-parallel, this is a reasonable assumption, and very useful from a theoretical perspective. With series-parallel graphs, we exhibit a polynomial algorithm on a simple architecture, and interesting NP-completeness proofs on more complicated ones. With general DAGs, we just would have an immediate and mean-

ingless NP-completeness proof on the simplest architecture. We choose to map such applications onto a current and promising platform: the chip multiprocessor. Those multi-core processors, whose cores are arranged along a bi-dimensional grid, are now really common, and are expected to stay the main architecture for many years.

Concerning the energy issue, we take into account both the energy consumed by the computations, and the energy consumed by the communications. We try to minimize the global energy, given a bound on the period of the application. After the complexity study, we present some heuristics and assess their performance through two sets of simulations. The first one is done on a randomly generated set of applications, whereas the second one evaluates the algorithms on a set of real-life applications.

Manhattan routing on chip multiprocessors [B1]

In this fifth chapter, we extend the work of the previous chapter, that was mainly focused on computations. Now we only consider the energy consumed by communications, still on a chip multiprocessor. We are given a set of communications, i.e., a set of triplets composed of a source core, a destination core, and a communication volume. Our aim is to minimize the energy consumed to route all those communications through the multi-core processor.

For the sake of simplicity, the implementation of the routing strategy in current chip multiprocessors is straightforward. Communications are following an XY route: data packets are sent horizontally first, then vertically. This may lead to a huge heterogeneity in the load of the links. And since the dissipated power is convex according to total volume of the communications going through a link, this power is not optimized at all. In this chapter, we show to which extent different strategies can impact the dissipated power. We compare XY routing, with Manhattan routings — where a communication can take any Manhattan path from the source core to the destination core — both single-path and multi-path. We dispose of an additional degree of freedom in this last case: a communication can be split among several Manhattan paths. We derive worst-case upper bounds of the ratio power consumed by an XY routing over power consumed by a Manhattan routing, and we exhibit examples realizing this ratio, hence showing the tightness of those bounds.

Assessment of bi-criteria heuristics for general directed acyclic graphs

In this last chapter, we aim at assessing the performance of often quoted heuristics for the following problem. We are given a set of dependent tasks, in the form of a Directed Acyclic Graph (DAG), which has already been mapped onto a set of processors running at their highest frequency, with the objective to minimize the total execution time. The goal of the algorithms we study is to minimize the energy consumption by downgrading processors while maintaining the makespan. We develop two variants of each heuristic, for both VDD-HOPPING and NO-VDD-HOPPING models. When VDD-HOPPING is allowed, a processor can be upgraded or downgraded at any time, while in the NO-VDD-HOPPING model, a task is associated to a unique speed.

We conduct a large set of simulations and vary several parameters: fatness of the graph, communication-to-computation ratio, number of possible frequencies and graph size. By inspecting the role of each parameter, we find domains in which heuristics are competitive or not. Finally we ask whether using VDD-HOPPING model achieves significant energy savings, compared to the NO-VDD-HOPPING model.

Chapter 1

On the performance of greedy algorithms for power consumption minimization

1.1 Introduction

In this chapter, we revisit the well-known greedy algorithm for scheduling independent jobs on parallel processors, with the objective of energy minimization. We assess the performance of the online version, as well as the performance of the offline version, which sorts the jobs by non-increasing size before execution.

For convenience, here is a quick background on the greedy algorithm for makespan minimization. Consider a set $\mathfrak{J}$ of n independent jobs $J_1, \ldots, J_n$ to be scheduled on a set $\mathfrak{P}$ of p parallel processors $\mathcal{P}_1, \ldots, \mathcal{P}_p$. Let a_i be the size of job J_i, that is the time it requires for execution. The algorithm comes in two versions, online and offline, or without/with sorting jobs. In the online version of the problem, jobs arrive on the fly. The ONLINE-GREEDY algorithm assigns the last incoming job to the currently least loaded processor. In the offline version of the problem (see [46]), all job sizes are known in advance, and the OFFLINE-GREEDY starts by sorting the jobs (largest sizes first). Then it assigns jobs to processors exactly as in the online version. The performance of both versions is characterized by the following propositions (see Figures 1.1 and 1.2 for an illustration of the worst-case scenarios):

Proposition 1.1. *For makespan minimization,* ONLINE-GREEDY *is a $2 - \frac{1}{p}$ approximation, and this approximation factor is met on the following instance:*
- $n = p(p-1) + 1$,
- $a_i = 1$ *for* $1 \leq i \leq n-1$,
- *and* $a_n = p$.

Proposition 1.2. *For makespan minimization,* OFFLINE-GREEDY *is a $\frac{4}{3} - \frac{1}{3p}$ approximation, and this approximation factor is met on the following instance:*
- $n = 2p + 1$,
- $a_{2i-1} = a_{2i} = 2p - i$ *for* $1 \leq i \leq p$,
- *and* $a_n = p$.

Assume that we can vary processor speeds, for instance through dynamic voltage scaling. In that case we can always use the smallest available speed for each processor, at the price of a dramatic decrease in performance.

ONLINE-GREEDY

$\mathcal{P}_1$	1	1	1	1	5
$\mathcal{P}_2$	1	1	1	1	
$\mathcal{P}_3$	1	1	1	1	
$\mathcal{P}_4$	1	1	1	1	
$\mathcal{P}_5$	1	1	1	1	

Optimal solution

$\mathcal{P}_1$	5				
$\mathcal{P}_2$	1	1	1	1	1
$\mathcal{P}_3$	1	1	1	1	1
$\mathcal{P}_4$	1	1	1	1	1
$\mathcal{P}_5$	1	1	1	1	1

Figure 1.1: Tight instance for ONLINE-GREEDY (with $p = 5$).

OFFLINE-GREEDY

$\mathcal{P}_1$	9	5	5
$\mathcal{P}_2$	9	5	
$\mathcal{P}_3$	8	6	
$\mathcal{P}_4$	8	6	
$\mathcal{P}_5$	7	7	

Optimal solution

$\mathcal{P}_1$	5	5	5
$\mathcal{P}_2$	9	6	
$\mathcal{P}_3$	9	6	
$\mathcal{P}_4$	8	7	
$\mathcal{P}_5$	8	7	

Figure 1.2: Tight instance for OFFLINE-GREEDY (with $p = 5$).

The problem is in fact a bi-criteria problem: given a bound M on the makespan, what is the schedule that minimizes the power consumption while enforcing the execution time bound?

For simplicity, we can assume that processors have continuous speeds (see [62, 40, 87, 95]), and scale the problem instance so that $M = 1$. This amounts to setting each processor speed equal to its workload, and to minimizing the total energy dissipated during an execution of length one time-unit. In other words, this amounts to minimizing the total dissipated power, which is proportional to the sum of the cubes of the processor speeds (a model commonly used, e.g. in [95, 24, 12, 30]).

Formally, let $alloc : \mathfrak{J} \rightarrow \mathfrak{P}$ denote the allocation function, and let $load(q) = \{i \mid alloc(J_i) = \mathcal{P}_q\}$ be the index set of jobs assigned to processor $\mathcal{P}_q$, for $1 \leq q \leq p$.

The power dissipated by $\mathcal{P}_q$ is $\left(\sum_{i \in load(q)} a_i\right)^3$, hence the objective is to minimize

$$\sum_{q=1}^{p} \left(\sum_{i \in load(q)} a_i \right)^3 . \tag{1.1}$$

This is to be contrasted with the makespan minimization objective, which writes

$$\max_{1 \le q \le p} \sum_{i \in load(q)} a_i \ . \tag{1.2}$$

However, because of the convexity of the cubic power function, the "natural" greedy algorithm is the same for both objectives: assigning the next job to the currently least loaded processor minimizes, among all possible assignments for that job, both the current makespan and dissipated power. We observe that when $p = 2$, the optimal solution is the same for both objectives. However, this is not true for larger values of p. For example, consider the instance with $n = 6$, $p = 3$, $a_1 = 8.1$, $a_2 = a_3 = 5$, $a_4 = a_5 = 4$ and $a_6 = 2$.

- The optimal solution for the makespan is the partition $\{J_1\}, \{J_2, J_3\}, \{J_4, J_5, J_6\}$, with makespan 10 and power 2531.441.
- The optimal solution for the power is the partition $\{J_1, J_6\}, \{J_2, J_4\}, \{J_3, J_5\}$, with makespan 10.1 (hence not optimal) and power 2488.301 (the processor loads are better balanced than in the previous solution, leading to a lower power consumption).

This example is illustrated in Figure 1.3.

Figure 1.3: Different optimal solutions for makespan and power minimization.

Just as the original makespan minimization problem, the (decision version of the) power minimization problem is NP-complete, and a PTAS (polynomial-time approximation scheme) can be derived. However, the greedy algorithm plays a key role in all situations where jobs arrive on the fly, or when the scheduling cost itself is critical. This was already true for the makespan problem, but may be even more important for the power problem, due to the environmental (or "green") computing perspective that applies to all application fields and computing platforms.

We discuss related work in Section 1.2. The main results of the chapter are summarized in Section 1.3, and compared to previously known results. Section 1.4 is devoted to a detailed proof of both theorems, and also we provide in Section 1.5 numerical values of the approximation factors for small values of p. We give some final remarks in Section 1.6.

1.2 Related work

The greedy algorithm has been widely studied in the literature, both in the offline and online versions. A more general problem than minimizing the sum of the cubes of the processor workloads (Equation (1.1)) is to minimize their L_r norm, i.e., the quantity

$$N_r = \left(\sum_{q=1}^{p} \left(\sum_{i \in load(q)} a_i \right)^r \right)^{\frac{1}{r}} . \quad (1.3)$$

Note that

$$N_\infty = \lim_{r \to \infty} N_r = \max_{1 \leq q \leq p} \sum_{i \in load(q)} a_i$$

is the makespan minimization objective (Equation (1.2)), while $(N_3)^3$ is the power minimization objective of this chapter (Equation (1.1)).

Chandra and Wong [23] consider the problem of minimizing N_2 in the offline version. They show that OFFLINE-GREEDY is a $\frac{5}{2\sqrt{6}}$ approximation algorithm for $r = 2$, but this bound is not tight: they give lower bounds for the approximation ratio of OFFLINE-GREEDY for the N_2 problem: their bound is $\frac{\sqrt{37}}{6}$ with an even number p of processors, $\frac{\sqrt{83}}{9}$ with $p = 3$ processors, and $\sqrt{\frac{37}{36} - \frac{1}{36p}}$ with an odd number $p \geq 5$ of processors. The gap between these bounds has been filled by Leung and Wei [79], who provide a tight approximation factor for the performance of OFFLINE-GREEDY for the N_2 problem.

Chandra and Wong [23] also provide lower and upper bounds for the approximation factor of OFFLINE-GREEDY for the general N_r problem. In particular for $r = 3$, their upper bound is $\frac{19}{45}\sqrt[3]{15} \approx 1.04$ (and their lower bound depends on the processor number p). Note that Theorem 1.2 below gives the exact approximation factor for any value of p, thereby closing the gap between lower and upper bounds. Finally, we point out that Chandra and Wong [23] do not deal with the online version of the problem, which Theorem 1.1 below completely solves.

Awerbuch et al. [11] discuss the problem of minimizing N_r for general r and for the online version of the problem. However, they have an additional rule: each job can be assigned only to a subset of the processors, called its *permissible* servers. They first study the problem with unit-size jobs (which is trivial without permissible servers), and they extend their analysis to the case where each job has a different execution cost on each of its admissible servers. They prove that ONLINE-GREEDY is a $1+\sqrt{2}$ approximation algorithm for $r = 2$, and a $\Theta(r)$ approximation algorithm in the general case.

Alon et al. [3] provide a PTAS (polynomial-time approximation scheme) to minimize N_r. This result is of great theoretical interest but only applies to the offline version of the problem, and is not related to the OFFLINE-GREEDY algorithm.

Finally, Avidor et al. [10] discuss the performance of ONLINE-GREEDY when minimizing N_r for general r. They provide an upper bound $2 - \Theta(\frac{\ln r}{r})$ for the approximation factor of ONLINE-GREEDY, independently of the number of processors. This is to be contrasted with Theorem 1.1 which provides a tight approximation factor for any processor number in the case $r = 3$.

1.3 Main contributions

The main results of the chapter are summarized in Theorems 1.1 and 1.2 below:

Theorem 1.1. *For power minimization,* ONLINE-GREEDY *is a* $f_p^{(\mathrm{on})}(\beta_p^{(\mathrm{on})})$ *approximation, where*

$$f_p^{(\mathrm{on})}(\beta) = \frac{\frac{1}{p^3}\left((1+(p-1)\beta)^3 + (p-1)(1-\beta)^3\right)}{\beta^3 + \frac{(1-\beta)^3}{(p-1)^2}},$$

and where $\beta_p^{(\mathrm{on})}$ *is the unique root in the interval* $[\frac{1}{p}, 1]$ *of the polynomial*

$$\begin{aligned} g_p^{(\mathrm{on})}(\beta) = {} & \beta^4(-p^3+4p^2-5p+2) + \beta^3(-2p^2+6p-4) \\ & + \beta^2(-4p+5) + \beta(2p-4) + 1. \end{aligned}$$

This approximation factor cannot be improved.

Theorem 1.2. *For power minimization,* OFFLINE-GREEDY *is a* $f_p^{(\mathrm{off})}(\beta_p^{(\mathrm{off})})$ *approximation, where*

$$f_p^{(\mathrm{off})}(\beta) = \frac{\frac{1}{p^3}\left(\left(1+\frac{(p-1)\beta}{3}\right)^3 + (p-1)\left(1-\frac{\beta}{3}\right)^3\right)}{\beta^3 + \frac{(1-\beta)^3}{(p-1)^2}},$$

and where $\beta_p^{(\mathrm{off})}$ *is the unique root in the interval* $[\frac{1}{p}, 1]$ *of the polynomial*

$$\begin{aligned} g_p^{(\mathrm{off})}(\beta) = {} & \beta^4(-9p^3+30p^2-27p+6) + \beta^3(-6p^2+18p-12) \\ & + \beta^2(-78p^2+126p+33) + \beta(18p-180) + 81. \end{aligned}$$

This approximation factor cannot be improved.

We point out that this chapter prove tight approximation factors for the problem of minimizing N_3, for any processor number p, both in the offline and online versions of the problem, which is totally new.

1.4 Proof of the main theorems

The proof of Theorems 1.1 and 1.2 is organized as follows:

- Proposition 1.3 provides a technical bound that is valid for both the online and offline versions;
- This technical bound is used in Proposition 1.4 to show that ONLINE-GREEDY is a $f_p^{(\mathrm{on})}(\beta_p^{(\mathrm{on})})$ approximation, and in Proposition 1.5 to show that OFFLINE-GREEDY is a $f_p^{(\mathrm{off})}(\beta_p^{(\mathrm{off})})$ approximation;
- Finally, instances showing that the above factors are tight are given in Proposition 1.6 for ONLINE-GREEDY, and in Proposition 1.7 for OFFLINE-GREEDY.

Proposition 1.3. *For any given instance, the performance ratio* $\frac{P_{\text{greedy}}}{P_{\text{opt}}}$ *of the greedy algorithm (*ONLINE-GREEDY *or* OFFLINE-GREEDY*) is such that*

$$\frac{P_{\text{greedy}}}{P_{\text{opt}}} \leq \frac{\left(\frac{S+(p-1)a_j}{p}\right)^3 + (p-1)\left(\frac{S-a_j}{p}\right)^3}{O^3 + (p-1)\left(\frac{S-O}{p-1}\right)^3}, \tag{1.4}$$

where

- P_{greedy} *is the power dissipated by the greedy algorithm;*
- P_{opt} *is the power dissipated in the optimal solution;*
- $S = \sum_{i=1}^{n} a_i$*;*
- O *is the largest processor load in the optimal solution;*
- j *is the index of the last job assigned to the processor that has the largest load in the greedy algorithm.*

Proof. For the optimal solution, we immediately have

$$P_{\text{opt}} \geq O^3 + (p-1)\left(\frac{S-O}{p-1}\right)^3 .$$

This is because of the definition of O, and of the convexity of the power function.

There remains to show that for the greedy algorithm,

$$P_{\text{greedy}} \leq \left(\frac{S+(p-1)a_j}{p}\right)^3 + (p-1)\left(\frac{S-a_j}{p}\right)^3 . \tag{1.5}$$

Without loss of generality, let $\mathcal{P}_1$ be the maximum loaded processor in the solution returned by the greedy algorithm. For all $q \in \{1, \dots, p\}$, let M_q be the load of processor $\mathcal{P}_q$ before the assignment of the job J_j, and let $u_q \geq 0$ be the sum of the sizes of all jobs assigned to $\mathcal{P}_q$ after J_{j-1}, as illustrated in Figure 1.4 for $p = 4$. By definition of j, we have $u_1 = a_j$. In the example, $u_3 = 0$, i.e., no jobs have been assigned to $\mathcal{P}_3$ after J_{j-1}.

The power returned by the greedy algorithm is thus:

$$P_{\text{greedy}} = \sum_{q=1}^{p}(M_q + u_q)^3 = (M_1 + a_j)^3 + \sum_{q=2}^{p}(M_q + u_q)^3 .$$

For $q \in \{2, \dots, p\}$, let v_q be the variation of the load of processor $\mathcal{P}_q$ from the average load of processors other than $\mathcal{P}_1$:

$$v_q = M_q + u_q - \frac{S - M_1 - a_j}{p-1} ,$$

and Figure 1.4 illustrates this notation. Then

$$P_{\text{greedy}} = \underbrace{(M_1 + a_j)^3 + \sum_{q=2}^{p}\left(\frac{S - M_1 - a_j}{p-1} + v_q\right)^3}_{f(M_1)} .$$

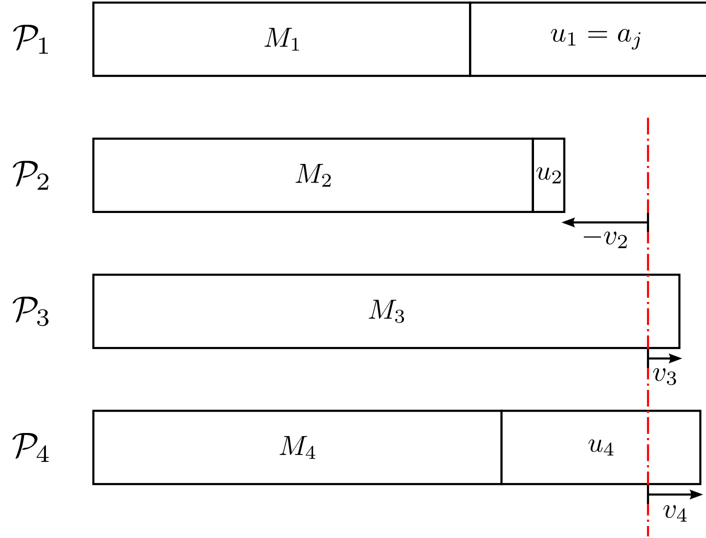


Figure 1.4: Notations for $p = 4$.

Note that the v_q can be either positive or negative (in the example, $v_2 \leq 0$ and $v_3 \geq 0$), and that their sum is always zero. To check this analytically, observe that $(M_1 + a_j) + \sum_{q=2}^{p} \left(\frac{S - M_1 - a_j}{p-1} + v_q \right) = S$, hence $\sum_{q=2}^{p} v_q = 0$.

Now, given the v_q, we have for $p \geq 2$, since $1 = \frac{p-1}{p-1}$:

$$f'(M_1) \geq \frac{3}{p-1} \times \sum_{q=2}^{p} \left((M_1 + a_j)^2 - \left(\frac{S - M_1 - a_j}{p-1} + v_q \right)^2 \right)$$

$$\geq \frac{3}{p-1} \times \sum_{q=2}^{p} \left(M_1 + a_j - \frac{S - M_1 - a_j}{p-1} - v_q \right) \times \left(M_1 + a_j + \frac{S - M_1 - a_j}{p-1} + v_q \right) .$$

By construction,

$$M_1 + a_j \geq \frac{S - M_1 - a_j}{p-1} + v_q,$$

therefore f is an increasing function.

Moreover, $\mathcal{P}_1$ is the least loaded processor before the assignment of J_j, thus a fortiori, for $q \in \{2, \ldots, p\}$,

$$M_1 \leq \frac{S - M_1 - a_j}{p-1} + v_q \ ,$$

hence

$$(p-1)M_1 \leq (S - M_1 - a_j) + \sum_{q=2}^{p} v_q = S - M_1 - a_j \ .$$

We derive that $M_1 \leq M_1^+$, where

$$M_1^+ = \frac{S - a_j}{p} \ . \tag{1.6}$$

Note that M_1^+ does not depend on the v_q. Since f is an increasing function, we have

$$P_{\text{greedy}} = f(M_1) \le f(M_1^+).$$

We had for $q \in \{2, \dots, p\}$, $M_1 \le \frac{S-M_1-a_j}{p-1} + v_q$, hence if $M_1 = M_1^+$,

$$v_q \ge \frac{p}{p-1} \times M_1^+ - \frac{1}{p-1} \times (S - a_j) = 0.$$

We deduce that, for $M_1 = M_1^+$ and $q \in \{2, \dots, p\}$, $v_q = 0$ (they are all nonnegative and their sum is null). Finally, we obtain

$$P_{\text{greedy}} \le f(M_1^+) = (M_1^+ + a_j)^3 + \frac{(S - a_j - M_1^+)^3}{(p-1)^2} ,$$

which, given the value of M_1^+ from Equation (1.6), directly leads to Equation (1.5). This concludes the proof. ∎

Proposition 1.4. *For power minimization,* ONLINE-GREEDY *is a* $f_p^{(\text{on})}(\beta_p^{(\text{on})})$ *approximation.*

Proof. We use the notations of Proposition 1.3. We first observe that $a_i \le O$, for all $i \in \{1, \dots, n\}$, by definition of O. In particular, $a_j \le O$.

We introduce $\beta = \frac{O}{S}$. Clearly, $\beta \in [\frac{1}{p}, 1]$, and we can rewrite Equation (1.4) as:

$$\frac{P_{\text{online}}}{P_{\text{opt}}} \le \underbrace{\frac{\frac{1}{p^3}\left(\left(1 + (p-1)\beta\right)^3 + (p-1)\left(1-\beta\right)^3\right)}{\beta^3 + \frac{(1-\beta)^3}{(p-1)^2}}}_{f_p^{(\text{on})}(\beta)} .$$

We now show that, for all p, $f_p^{(\text{on})}$ has a single maximum in $[\frac{1}{p}, 1]$. After differentiating with respect to β and eliminating some positive multiplicative factor, we obtain that the sign of $\left(f_p^{(\text{on})}\right)'$ is that of $g_p^{(\text{on})}$, where:

$$g_p^{(\text{on})}(\beta) = \beta^4(-p^3 + 4p^2 - 5p + 2) + \beta^3(-2p^2 + 6p - 4) + \beta^2(-4p + 5) + \beta(2p - 4) + 1.$$

Differentiating again two times, we obtain:

$$\left(g_p^{(\text{on})}\right)'(\beta) = 4\beta^3(-p^3 + 4p^2 - 5p + 2) + 3\beta^2(-2p^2 + 6p - 4) + 2\beta(-4p + 5) + 2p - 4;$$

$$\left(g_p^{(\text{on})}\right)''(\beta) = 24\beta^2 - 24\beta + 10 - 8p + p(-12\beta p + 36\beta - 60\beta^2) + 48p^2\beta^2 - 12p^3\beta^2 .$$

If $p \ge 5$,

$$\left(g_p^{(\text{on})}\right)''(\beta) \le 34 - 40 + p(-60 + 36) + p^2(-60 + 48)\beta^2 \le 0 .$$

We check that

$$\left(g_2^{(\mathrm{on})}\right)''(\beta) = -6 \le 0\ ,$$
$$\left(g_3^{(\mathrm{on})}\right)''(\beta) = -24\beta - 14 - 48\beta^2 \le 0 \ \text{ and}$$
$$\left(g_4^{(\mathrm{on})}\right)''(\beta) = -72\beta - 22 - 216\beta^2 \le 0\ ,$$

hence $\left(g_p^{(\mathrm{on})}\right)'$ is a decreasing function for all $p \ge 2$ in the interval $[\frac{1}{p}, 1]$.

Next, we show that

$$\left(g_p^{(\mathrm{on})}\right)'(1) = -4p^3 + 10p^2 - 8p + 2 \le 0,$$

and hence either $g_p^{(\mathrm{on})}$ is increasing and then decreasing in the interval $[\frac{1}{p}, 1]$, or $g_p^{(\mathrm{on})}$ is decreasing in the whole interval. Indeed, for $p = 2$, $\left(g_2^{(\mathrm{on})}\right)'(1) = -6 \le 0$, and for $p \ge 3$,

$$\left(g_p^{(\mathrm{on})}\right)'(1) \le p^2(-12 + 10) - 24 + 2 \le 0.$$

We now check the values of $g_p^{(\mathrm{on})}$ at the interval bounds: for $p \ge 2$, we have

$$g_p^{(\mathrm{on})}(1) = -p + 2p^2 - p^3 \le 0\ ,\ \text{and}$$

$$g_p^{(\mathrm{on})}\left(\frac{1}{p}\right) = 3 - 11/p + 15/p^2 - 9/p^3 + 2/p^4 \ge 0,$$

since $g_2^{(\mathrm{on})}\left(\frac{1}{2}\right) = \frac{1}{4}$, $g_3^{(\mathrm{on})}\left(\frac{1}{3}\right) = \frac{56}{81}$, and for all $p \ge 4$, $g_p^{(\mathrm{on})}\left(\frac{1}{p}\right) \ge 3 - 11/p \ge \frac{12-11}{p} \ge 0$.

In both cases (either $g_p^{(\mathrm{on})}$ is increasing then decreasing, or $g_p^{(\mathrm{on})}$ is only decreasing), since

$$g_p^{(\mathrm{on})}\left(\frac{1}{p}\right) \ge 0 \quad \text{and} \quad g_p^{(\mathrm{on})}(1) \le 0,$$

we conclude that $g_p^{(\mathrm{on})}$ has a single zero $\beta_p^{(\mathrm{on})}$ in $[\frac{1}{p}, 1]$, for which $f_p^{(\mathrm{on})}$ attains its maximum. Finally ONLINE-GREEDY is a $f_p^{(\mathrm{on})}(\beta_p^{(\mathrm{on})})$ approximation. ■

Proposition 1.5. *For power minimization,* OFFLINE-GREEDY *is a $f_p^{(\mathrm{off})}(\beta_p^{(\mathrm{off})})$ approximation.*

Proof. We follow the same line of reasoning as in Proposition 1.4, with $O = \beta S$, but we now further assume that $a_j \le O/3$. Indeed, if $a_j > O/3$, there are at most two jobs assigned to each processor in the optimal solution. But then $n \le 2p$, and for all such instances OFFLINE-GREEDY is optimal (this is the same argument as for the makespan minimization problem, due to the convexity of the power function). With $a_j \le O/3 = \beta S/3$, we rewrite Equation (1.4) as:

$$\frac{P_{\text{offline}}}{P_{\text{opt}}} \le \underbrace{\frac{\frac{1}{p^3}\left(\left(1 + \frac{(p-1)\beta}{3}\right)^3 + (p-1)\left(1 - \frac{\beta}{3}\right)^3\right)}{\beta^3 + \frac{(1-\beta)^3}{(p-1)^2}}}_{f_p^{(\mathrm{off})}(\beta)}\ .$$

The sign of $\left(f_p^{(\text{off})}\right)'$ is the sign of $g_p^{(\text{off})}$, where:

$$\begin{aligned} g_p^{(\text{off})}(\beta) = \quad & \beta^4(-9p^3+30p^2-27p+6)+\beta^3(-6p^2+18p-12) \\ & +\beta^2(-78p^2+126p+33)+\beta(18p-180)+81. \end{aligned}$$

Differentiating again two times, we obtain:

$$\begin{aligned} \left(g_p^{(\text{off})}\right)'(\beta) = \quad & 4\beta^3(-9p^3+30p^2-27p+6) \\ & +3\beta^2(-6p^2+18p-12)+2\beta(-78p^2+126p+33)+18p-180\ ; \end{aligned}$$

$$\begin{aligned} \left(g_p^{(\text{off})}\right)''(\beta) = \quad & 12\beta^2(-9p^3+30p^2-27p+6)+6\beta(-6p^2+18p-12) \\ & -156p^2+252p+66\ . \end{aligned}$$

If $p \geq 4$,

$$\begin{aligned} \left(g_p^{(\text{off})}\right)''(\beta) \leq \quad & 12\beta^2((-36+30)p^2-108+6)+6\beta((-24+18)p-12) \\ & +(-588+252)p+(-80+66) \\ \left(g_p^{(\text{off})}\right)''(\beta) \leq \quad & 0. \end{aligned}$$

Now $\left(g_2^{(\text{off})}\right)''(\beta) = -54$ and

$$\left(g_3^{(\text{off})}\right)''(\beta) = -576\beta^2-72\beta-582 \leq 0,$$

thus for all $p > 1$ and $\frac{1}{p} \leq \beta \leq 1$,

$$\left(g_p^{(\text{off})}\right)''(\beta) \leq 0.$$

Therefore $g_p^{(\text{off})}$ is concave.

Let us now check the values of $g_p^{(\text{off})}$ at the interval bounds. We have

$$g_p^{(\text{off})}\left(\frac{1}{p}\right) \geq 21-35+15 \geq 0\ ,\ \text{and}$$

$$g_p^{(\text{off})}(1) = -9p^3-54p^2+135p-72 \leq 0\ ,$$

since $g_2^{(\text{off})}(1) = -72-216+270-72 \leq 0$, and for $p \geq 3$, $g_p^{(\text{off})}(1) \leq p(-27-162+135)-72 \leq 0$.

We conclude that for all $p > 1$, $f_p^{(\text{off})}$ has a single maximum in $[\frac{1}{p}, 1]$, reached for $\beta = \beta_p^{(\text{off})}$, where $g_p^{(\text{off})}(\beta_p^{(\text{off})}) = 0$. Finally OFFLINE-GREEDY is a $f_p^{(\text{off})}(\beta_p^{(\text{off})})$ approximation. ■

Proposition 1.6. *The approximation factor $f_p^{(\mathrm{on})}(\beta_p^{(\mathrm{on})})$ for* ONLINE-GREEDY *cannot be improved.*

Proof. Consider an instance with p processors and $n = p(p-1)+1$ jobs, where for all $i \in \{1, \ldots, n-1\}$, $a_i = 1$, and $a_n = B = \frac{\beta_p^{(\mathrm{on})} p(p-1)}{1-\beta_p^{(\mathrm{on})}}$.

ONLINE-GREEDY assigns $p-1$ unit-size jobs to each processor, and then the big job is assigned to any processor, leading to a power dissipation of:

$$P_{\mathrm{online}} = \left(\frac{S + (p-1)a_j}{p}\right)^3 + (p-1)\left(\frac{S - a_j}{p}\right)^3 ,$$

where $j = n$.

From $\beta_p^{(\mathrm{on})} \geq \frac{1}{p}$, we deduce that $B \geq p$. Therefore the optimal solution assigns J_n to the first processor, and p unit-size jobs to each other processor. We have $a_j = O = B$ and for $q \in \{2, \ldots, p\}$,

$$\sum_{i \in load(q)} a_i = p = \frac{S - O}{p-1}, \text{ and hence}$$

$$P_{\mathrm{opt}} = O^3 + (p-1)\left(\frac{S-O}{p-1}\right)^3 .$$

Moreover we have $O = \beta_p^{(\mathrm{on})} S$:

$$\begin{aligned} O - \beta_p^{(\mathrm{on})} S &= B - \beta_p^{(\mathrm{on})}(B + p(p-1)) \\ &= B - p(p-1)\beta_p^{(\mathrm{on})}\left(\frac{\beta_p^{(\mathrm{on})}}{1-\beta_p^{(\mathrm{on})}} + 1\right) \\ O - \beta_p^{(\mathrm{on})} S &= 0 . \end{aligned}$$

Therefore, for this instance,

$$\frac{P_{\mathrm{online}}}{P_{\mathrm{opt}}} = f_p^{(\mathrm{on})}(\beta_p^{(\mathrm{on})}) ,$$

which concludes the proof. ■

Proposition 1.7. *The approximation factor $f_p^{(\mathrm{off})}(\beta_p^{(\mathrm{off})})$ for the ratio of the* OFFLINE-GREEDY *cannot be improved.*

Proof. Consider an instance with p processors and $n = 2p+1$ jobs, where for all $1 \leq i \leq p$,

$$a_{2i-1} = a_{2i} = 2p - i + v_i,$$

and where $a_n = p + v_p$.

We define

$$A = \frac{3p(1 - \beta_p^{(\mathrm{off})} p)}{\beta_p^{(\mathrm{off})}(p+1) - 3}, \text{ and}$$

$$\forall 1 \leq i \leq p, \quad v_i = \frac{i-1}{p-1} A.$$

We first show that the jobs are sorted in non-increasing order:
– For $1 \leq i \leq p$, $a_{2i-1} = a_{2i}$;
– $a_n = a_{n-1}$ $(= a_{2p})$;
– For $1 \leq i \leq p-1$,
$a_{2i+1} - a_{2i} = -1 + v_{i+1} - v_i = -1 + \frac{A}{p-1}$.
Consider the function

$$\hat{h}_p : \beta \mapsto \frac{3p(\beta p - 1)}{3 - \beta(p+1)}.$$

Its derivative is nonnegative, hence $\hat{h}_p$ is increasing.

We now prove that $\beta_p^{(\text{off})} \leq 3/(2p+1)$, which ensures that

$$A = \hat{h}_p(\beta_p^{(\text{off})}) \leq \hat{h}_p(3/(2p+1)) = p - 1,$$

and therefore $a_{2i+1} - a_{2i} = \frac{A}{p-1} - 1 \leq 0$. Recall that $\beta_p^{(\text{off})}$ is the unique root in the interval $[\frac{1}{p}, 1]$ of $g_p^{(\text{off})}$ (see Theorem 1.2). We already know that $g_p^{(\text{off})}(\frac{1}{p}) \geq 0$ (see proof of Proposition 1.5). We now prove that

$$g_p^{(\text{off})}(3/(2p+1)) \leq 0\ .$$

Indeed, we have

$$g_p^{(\text{off})}(3/(2p+1)) = -\frac{135p(8p^3 + 3p^2 - 30p + 19)}{(2p+1)^4},$$

$$\text{and } 8p^3 + 3p^2 - 30p + 19 \geq p(32 - 30) + 19 \geq 0.$$

Therefore, $\beta_p^{(\text{off})} \leq \frac{3}{2p+1}$, which proves that $a_{2i+1} \leq a_{2i}$, and hence the jobs are sorted in non-increasing order.

Before the assignment of the last job, all processor loads are perfectly balanced. OFFLINE-GREEDY first assigns $J_1, J_2, \ldots, J_p$ to $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p$ respectively. Then it assigns the jobs $J_{p+1}, J_{p+2}, \ldots, J_{2p}$ to $\mathcal{P}_p, \mathcal{P}_{p-1}, \ldots, \mathcal{P}_1$ respectively. After these assignments, for all $i \in \{1, \ldots, \lceil p/2 \rceil\}$, the load of processor $\mathcal{P}_{2i-1}$ is:

$$\begin{aligned} a_{2i-1} + a_{2(p-(i-1))} &= 3p + v_i + v_{p-(i-1)} \\ &= 3p + \frac{i-1}{p-1}A + \frac{p-i}{p-1}A \\ a_{2i-1} + a_{2(p-(i-1))} &= 3p + A. \end{aligned}$$

Moreover, for all $i \in \{1, \ldots, \lfloor p/2 \rfloor\}$, the load of processor $\mathcal{P}_{2i}$ is

$$a_{2i} + a_{2(p-(i-1))-1} = a_{2i-1} + a_{2(p-(i-1))} = 3p + A.$$

The last job J_n is assigned to any processor, and the power dissipated by OFFLINE-GREEDY is:

$$P_{\text{offline}} = \left(\frac{S + (p-1)a_j}{p}\right)^3 + (p-1)\left(\frac{S - a_j}{p}\right)^3,$$

where $j = n$.

The optimal solution assigns $J_1, J_2, \ldots, J_{p-1}$ to $\mathcal{P}_2, \mathcal{P}_3, \ldots, \mathcal{P}_p$ respectively. It assigns the jobs $J_p, J_{p+1}, \ldots, J_{2p-2}$ to $\mathcal{P}_p, \mathcal{P}_{p-1}, \ldots, \mathcal{P}_2$ respectively. The last three jobs J_{2p-1}, J_{2p} and J_{2p+1} are assigned to $\mathcal{P}_1$, which is the most loaded processor.

The loads of processors $\mathcal{P}_2, \mathcal{P}_3, \dots, \mathcal{P}_p$ are perfectly balanced in the optimal assignment, and their load is $3p + pA/(p-1)$:

– For all $i \in \{1, \dots, \lfloor p/2 \rfloor\}$, the load of processor $\mathcal{P}_{2i}$ is

$$\begin{aligned} a_{2i-1} + a_{2(p-i)} &= 3p + v_i + v_{p-i} \\ &= 3p + \frac{i-1}{p-1}A + \frac{p-i+1}{p-1}A \\ a_{2i-1} + a_{2(p-i)} &= 3p + pA/(p-1). \end{aligned}$$

– For all $i \in \{1, \dots, \lceil p/2 \rceil - 1\}$, the load of processor $\mathcal{P}_{2i+1}$ is

$$\begin{aligned} a_{2i} + a_{2(p-i)-1} &= a_{2i-1} + a_{2(p-i)} \\ a_{2i} + a_{2(p-i)-1} &= 3p + pA/(p-1). \end{aligned}$$

Finally, the load of processor $\mathcal{P}_1$ is

$$O = 3a_n = 3p + 3A \ ,$$

and since $3p + 3A \geq 3p + pA/(p-1)$, it is the most loaded processor.

We can then compute the corresponding power consumption. Note that the total load $S - O$ is equally divided between $p-1$ processors, and hence $3p + pA/(p-1) = \frac{S-O}{p-1}$. We obtain:

$$P_{\text{opt}} = O^3 + (p-1)\left(\frac{S-O}{p-1}\right)^3 \ .$$

To conclude the proof, we need to prove that

$$O = \beta_p^{(\text{off})} S \ .$$

Note that

$$\begin{aligned} S &= 3p^2 + v_p + 2\sum_{i=1}^{p} v_i = 3p^2 + A + \frac{2A}{p-1}\sum_{i=0}^{p-1} i \\ S &= 3p^2 + (p+1)A \ , \end{aligned}$$

and therefore

$$\begin{aligned} \beta_p^{(\text{off})} S - O &= 3p(\beta_p^{(\text{off})} p - 1) + (\beta_p^{(\text{off})}(p+1) - 3)A \\ &= 3p(\beta_p^{(\text{off})} p - 1) + 3p(\beta_p^{(\text{off})} p - 1) \\ &= 0, \end{aligned}$$

which leads to the desired result.

Finally, since $a_j = a_n = O/3$, we can easily verify that the ratio of this instance is

$$\frac{P_{\text{offline}}}{P_{\text{opt}}} = f_p^{(\text{off})}(\beta_p^{(\text{off})}) \ .$$

■

1.5 The approximation factor as a function of p

We provide in this section a few observations on the values of the approximation factor of ONLINE-GREEDY and OFFLINE-GREEDY for large values of p. Using Taylor expansions, we derive the following asymptotic values for large p:

- For large p, $\beta_p^{(\text{on})} = \left(\frac{2}{p^2}\right)^{1/3} + O(1/p)$. Note that $\sqrt[3]{2} \approx 1.260$.
- For large p, $\beta_p^{(\text{off})} = \frac{3(1+\sqrt{79})}{26p} + O(1/p^2)$. Note that $\frac{3(1+\sqrt{79})}{26} \approx 1.141$.

It is worth pointing out that both ONLINE-GREEDY and OFFLINE-GREEDY are asymptotically optimal when p is large, while in the case of makespan minimization, the asymptotic approximation factor of ONLINE-GREEDY was equal to 2 and that of OFFLINE-GREEDY equal to $4/3$.

For $p = 2$ we have exact values:

$$\beta_2^{(\text{on})} = \frac{\sqrt{3}}{3} \quad \text{and} \quad f_2^{(\text{on})}(\beta_2^{(\text{on})}) = 1 + \frac{\sqrt{3}}{2} \approx 1.866,$$

$$\text{while} \quad \beta_2^{(\text{off})} = \frac{\sqrt{91}-8}{3} \quad \text{and} \quad f_2^{(\text{off})}(\beta_2^{(\text{off})}) = 1 + \frac{\sqrt{91}+10}{18} \approx 1.086.$$

We report representative numerical values in Table 1.1. We observe that ONLINE-GREEDY is at most 50% more costly than the optimal for $p \geq 64$, while OFFLINE-GREEDY always remains within 10% of the optimal, and gets within 5% for $p \geq 7$.

	ONLINE-GREEDY		OFFLINE-GREEDY	
p	$\beta_p^{(\text{on})}$	$f_p^{(\text{on})}(\beta_p^{(\text{on})})$	$\beta_p^{(\text{off})}$	$f_p^{(\text{off})}(\beta_p^{(\text{off})})$
2	0.577	1.866	0.513	1.086
3	0.444	2.008	0.350	1.081
4	0.372	2.021	0.267	1.070
5	0.325	2.001	0.216	1.061
6	0.292	1.973	0.181	1.054
7	0.266	1.943	0.156	1.048
8	0.246	1.915	0.137	1.043
64	0.0696	1.461	0.0177	1.006
512	0.0186	1.217	0.00223	1.00083
2048	0.00479	1.104	0.000278	1.00010
2^{24}	0.0000192	1.006	0.0000000680	1.000000025

Table 1.1: Values for the approximation factors of ONLINE-GREEDY and OFFLINE-GREEDY.

1.6 Conclusion

In this chapter, we have fully characterized the performance of the greedy algorithm for the power minimization problem. We have provided tight approximation factors for any processor number p, both in the offline and online versions of the problem. These results extend those of a long series of papers, and completely solve the N_3 minimization problem.

On the practical side, further work could be devoted to conducting experiments with a more complicated power model, that would include static power in addition to dynamic power (see for example the model for the Intel Xscale [60], detailed in [31, 28, 89]). With such a model, the "natural" greedy algorithm would assign the next job to the processor that minimizes the increment in total power. There would then be two choices, either the currently least loaded processor, or a currently unused processor (at the price of more static power to be paid).

Chapter 2

Mapping concurrent streaming applications

2.1 Introduction

In this chapter, we complicate the problem addressed in the previous chapter by adding dependencies between tasks we have to schedule and by including a new performance-related criterion in the optimization objectives. Moreover, instead of finding theoretical bounds on existing algorithms, the goal is to study the problem complexity. For the NP-complete instances, we also design new heuristics, based on polynomial-time optimal algorithm of simpler problems, and we experimentally evaluate their performance.

We aim at optimizing the parallel execution of several pipelined applications that execute concurrently on a given platform. We focus in this work on pipelined applications whose structure is a linear chain of tasks. Such applications are ubiquitous in streaming environments, as for instance video and audio encoding and decoding, DSP applications, image processing, and so on [39, 113, 55, 122, 123]. Furthermore, the regularity of these applications render them amenable to a high-level parallel programming approach based on algorithmic skeletons [37, 97]. Skeletons ease the task of the application developer and make it easy to tailor his/her specific problem to a target platform.

In a linear pipelined application, a series of data sets enters the input stage and progresses from stage to stage until the final result is computed. Each stage corresponds to a distinct task and has its own communication and computation requirements: it reads an input from the previous stage, processes the data and outputs a result to the next stage. Each data set is input to the first stage, and final results are output from the last stage. The pipeline operates in synchronous mode: after a transient behavior due to the initialization delay, a new data set is completed every period. Mapping such applications onto parallel platforms is a challenging problem, that becomes even more difficult when platforms are heterogeneous (nowadays a standard assumption). Another level of difficulty is added when considering several independent applications which are executed concurrently on the platform and that compete for available resources.

The objective is to minimize the energy consumption of the whole platform, while satisfying given performance-related bounds on the period and latency of each application. The multi-criteria approach targets a trade-off between the users and the platform manager. The formers have specific requirements for their applications, while the latter has crucial economical and environmental constraints.

The main performance-oriented criteria for pipelined applications are period and latency [110, 111, 117, 118, 17, 18, 122]. The period of an application is the inverse of the throughput, i.e., it corresponds to the time interval between the arrival of two consecutive data sets. The period is dictated by the critical resource: it is equal to the longest cycle time of a processor. For instance under a strict one-port communication model with no overlap of communications and computations, it is the sum of the time to

perform all incoming communications, the time to perform all outgoing communications, and the total computation time. With overlap, we simply replace the sum of these three terms by their maximum. In some cases, the period is fixed by the applicative setting, and we must ensure that data sets are processed fast enough so that there is no accumulation of data sets in the pipeline. The latency of an application is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. In classical scheduling, for non-streaming applications, the latency is the makespan, or exeuction time (as in previous chapter). For streaming applications, there are several approaches to compute the latency. The most accurate model is the PATH model, in which the latency is computed as the length of the path taken by any data set. With the WAVEFRONT model, we rather consider that each data set progresses concurrently within a period, and the latency is then a multiple of the period, as suggested by Hary and Özgüner in [55].

The two performance criteria alone already are antagonistic. The smallest latency is obtained when no communication occurs, i.e., when the same (fastest) processor executes all the stages of an application. However, such a mapping may well exceed the bound on the period, since the same processor must process an entire application. Moreover, when several applications run concurrently, the scheduler must decide which resources to select and assign to each application, so that all users receive a fair share of the platform.

Adding energy consumption as a third criterion renders everything even more complex. Obviously, energy is minimized by enrolling a single processor for all applications, namely the one with the smallest speed available; but such a mapping would most certainly exceed period and latency bounds.

Our goal is to execute all applications efficiently while minimizing the energy consumed. Unfortunately, the goals of low power consumption and efficient scheduling are contradictory. Indeed, period and/or latency can be minimized by using more energy to speed up processors, while energy can be minimized by reducing processor speeds, hence performance-related objectives. How to deal with these contradictory objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to be optimized. But is it natural for the user to maximize the quantity $0.7T + 0.3E$, where T is the period and E the energy? Since criteria are very different in nature, it does not make much sense for a user to make a linear combination of them. Thus we advocate the use of multi-criteria mappings with thresholds. Now, each criteria combination can be handled in a natural and meaningful way: one single criterion is optimized, under the condition that a threshold is enforced for all other criteria. This leads to two interesting questions. If we fix energy, we get the *laptop problem*, which asks "What is the best schedule achievable using a particular energy budget, before battery becomes critically low?" Fixing schedule quality gives the *server problem*, which asks "What is the least energy required to achieve a desired level of performance?"

The optimization problem can then be stated as follows: given a set of applications and a computational platform, which stage to assign to which processor? We consider three different mapping strategies: *one-to-one* mappings, for which each application stage is allocated to a distinct processor; *interval* mappings, where each participating processor is assigned an interval of consecutive stages of the same application; and *general* mappings which are fully arbitrary. These mapping strategies have been widely used in the literature when mapping one single application (see [110, 111, 17]), and we extend them naturally to map of several concurrent applications.

We target three different platform types: *fully homogeneous* platforms have identical processors and interconnection links; *communication homogeneous* platforms have identical links but different-speed processors, thus introducing a first degree of heterogeneity; and finally, *fully heterogeneous* platforms, with different-speed processors and different capacity links, constitute the most difficult problem instance.

The chapter is organized as follows. We first review related work in Section 2.2. Then, we illustrate and motivate the problem with a simple example in Section 2.3. The framework is described in Section 2.4. The next two sections constitute the heart of the chapter: we assess the complexity of all problem instances. In Section 2.5, we establish the complexity of mapping problems with the PATH latency model, while we investigate the complexity with the WAVEFRONT latency model in Section 2.6. In Section 2.7, we study the relative and absolute performance, with respect to an optimal integer linear program, of heuristics that we designed, under the WAVEFRONT latency model. We conclude in Section 2.8.

2.2 Related work

The problem of mapping a single linear chain application onto parallel platforms in order to minimize latency and/or period has already been widely studied, in particular on homogeneous platforms (see the pioneering papers [110] and [111]) and later for heterogeneous platforms (see [17, 18]), considering the PATH latency model. These results focus on the mapping of one single application, while we add the complexity of satisfying several users who each have different requirements for their applications. We were able to extend polynomial time algorithms to this multi-application setting, and to exhibit cases in which the problem becomes NP-hard because of this additional difficulty. Of course, problem instances which were already NP-hard with a single application remain difficult with several concurrent applications.

Moreover, we consider a new and important objective function, namely energy minimization, and this is the first study (to the best of our knowledge) which combines performance-related objectives with energy in the context of pipelined applications. As expected, combining all three criteria (period, latency and energy) leads to even more difficult optimization problems: the problem is NP-hard even with a single application on a fully homogeneous platform (for interval mappings with the PATH latency model).

In order to adjust energy consumption, we use the Dynamic Voltage Scaling (DVS) technique. DVS has been extensively studied in several papers, for mapping onto a single-core processor, a multi-core processor, or a set of processors.

Slack reclamation techniques are used for frame-based hard real-time embedded system in [125]: a set of independent tasks, provided with their WCEC (Worst Case Execution Cycle) and sharing a common deadline, has to be mapped onto a processor. If a task needs less cycles than its WCEC, the dynamically obtained slack allows the processor to run at a lower frequency and therefore to spare energy. This work is extended in [126], where the energy model includes time and energy penalties when the processor frequency is changing. Those transition overheads are also taken into account in [6], but tasks are interdependent.

Then [35] maps applications which consists of a program modeled with a sequential part and another part which can be parallel, onto a multi-core processor. Bunde [22] focuses on the problem of offline scheduling unit time tasks with release dates, while minimizing the makespan or the total flow time on one processor. He extends this work from one processor to multi-processors.

Authors in [32] study the problem of scheduling real-time tasks on two heterogeneous processors. They provide a FPTAS to derive a solution very close to the optimal energy consumption with a reasonable complexity, while in [50], the authors design heuristics to map a set of dependent tasks with deadlines onto a set of homogeneous processors, with the possibility of changing a processor speed during the execution of a task. [58] proposes a greedy algorithm based on affinity to assign frame-based real-time tasks, and then they re-assign them in pseudo-polynomial time when any processing speed can

be assigned for a processor. In [40], leakage energy is the focus for mapping applications represented as DAGs. In [115], the authors are interested about scheduling task graphs with data dependencies while minimizing the energy consumption of both the processors and the inter-processor communication devices, while assuming the communication times are negligible compared to the computation times.

All these problems are quite different from ours, since we focus on pipelined applications of infinite duration, thus considering power instead of total energy consumption. Due to the streaming nature of the applications, we do not allow for changing the processor speed during execution.

2.3 Motivating example

In this example, we have two applications and three processors, as shown on Figure 2.1. The first stage of App_1 computes 3 operations, and then sends a data of size 3 to the second stage; the second stage first receives a data of size 3, then computes 2 operations, and finally sends a data of size 1, and so on. If both stages are assigned to the same processor, there is no communication cost to pay; otherwise this cost depends on the communication volume (3 between the first and the second stage in this case), and on the link bandwidth between the corresponding processor pair. All communication link bandwidths are set to 1. For the computational platform, each processor has two execution modes. For instance, $\mathcal{P}_1$ can process 3 operations per time unit in its first mode, and 6 in its second one, against 6 or 8 for $\mathcal{P}_2$, and 1 or 6 for $\mathcal{P}_3$.

We compute the global period as follows: $T = \max(T_1, T_2)$, where T_i is the period of the ith application ($i = 1, 2$). The global latency is defined in a similar way, as the maximum of the latency achieved by all applications. Like in the previous chapter, the energy consumption of a processor is equal to the cube of its speed (see Section 2.4.5 for more details on the model for energy consumption). Note that when the energy is not a criterion to minimize, all processors can run in their higher modes (as fast as possible), because this can only improve the performance-related criteria (period and latency). In this case, either a processor is used at its fastest speed, or it is turned off.

2.3.1 Interval mappings

First we restrict to interval mappings, where a processor can be assigned only a set of consecutive stages of a single application.

In order to minimize the period without energy constraints, we map the whole first application onto processor $\mathcal{P}_3$, the first half of the second application onto processor $\mathcal{P}_2$, and the rest onto processor $\mathcal{P}_1$. The period is then:

$$\max\left(\frac{3+2+1}{6}, \max\left(\frac{2+6}{8}, \frac{1}{1}, \frac{4+2}{6}\right)\right) = 1 \ . \tag{2.1}$$

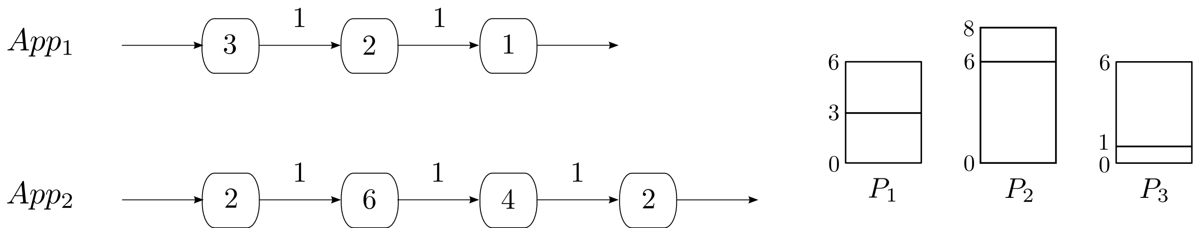


Figure 2.1: Example with two applications and three multi-modal processors.

Equation (2.1) reads as follows: we compute the cycle-time of each processor as the maximum time spent for incoming communications, computations, and outgoing communications, thus considering a model in which communications and computations are overlapped. We then take the maximum of these quantities to derive the period. There is only one communication to pay in the second application since it is split between two processors. Note that the cycle-time of each processor is exactly 1 and there is no idle time on computation, thus it is not possible to achieve a better period: this mapping is optimal for the period minimization problem.

The minimum latency is obtained by removing all communications and using the fastest processors. A mapping that returns the optimal latency (in the absence of other criteria) is for instance the one which maps the first application on $\mathcal{P}_1$ and the second application on $\mathcal{P}_2$, thus achieving a global latency of:

$$\max\left(\frac{3+2+1}{6}, \frac{2+6+4+2}{8}\right) = 1.75\ . \tag{2.2}$$

In Equation (2.2), we simply compute the longest execution path for each application following the PATH latency model. The period of each application is, in this case, equal to its latency, and the WAVEFRONT model returns the same latency (one single period for the execution of a data set). The bottleneck is the second application, and we cannot achieve a better latency since we pay no communication and use the fastest processor for this application. This latency is thus optimal.

The minimum energy is obtained when we use fewer processors, each running in its slowest mode. Since we assume that a processor cannot be assigned stages of two different applications, two processors are required in the example. For instance, we can map the first application on $\mathcal{P}_1$ running in its lowest mode and the second application on $\mathcal{P}_3$ running in its lowest mode too, thus achieving an energy of $3^3 + 1^3 = 28$. This is the minimum energy consumption required to run both applications. We observe that the period is then:

$$\max\left(\frac{3+2+1}{3}, \frac{2+6+4+2}{1}\right) = 14\ . \tag{2.3}$$

As expected, running at a slower mode to save energy leads to poorer performances. Trade-offs must be found when considering several antagonistic optimization criteria.

For instance, if we try to minimize the energy consumption under the constraint that the period is not greater than 2, we can use the first mode of each processor. Then the first application is mapped onto $\mathcal{P}_1$, the first three stages of the second application are mapped onto $\mathcal{P}_2$ and its last stage is mapped onto $\mathcal{P}_3$. The global period is 2, and the consumed energy is $3^3 + 6^3 + 1^3 = 244$. This may be quite a reasonable compromise between energy and period: indeed, with the mapping minimizing the period (period of 1), the energy consumption was $6^3 + 8^3 + 6^3 = 944$. With this mapping, the latency model impacts the result. With the PATH model, we compute the longest path followed by a data set, it is $\max\left(\frac{3+2+1}{3}, \frac{2+6+4}{6} + \frac{1}{1} + \frac{2}{1}\right) = 5$, while with the WAVEFRONT model, it takes three periods for a data set to be computed by the second application, leading to a latency of $3 \times 2 = 6$.

2.3.2 General mappings

With general mappings, it is possible to assign any subset of stages to the processors. For instance, we consider the mapping in which the first stage of application one, and the second and third stages of application two, are all mapped onto the second processor, running at speed 6. The other stages are mapped onto the first processor, running at speed 3.

The energy consumption is then $3^3 + 6^3 = 243$. For the period, we take the maximum between the

periods of both processors, accounting both for computation and communication costs:

$$\max\left(\frac{1+1}{1}, \frac{2+1+2+2}{3}, \frac{1}{1}, \frac{3+6+4}{6}, \frac{1+1}{1}\right) = \frac{7}{3} . \tag{2.4}$$

Note that there are two communications from $\mathcal{P}_2$ to $\mathcal{P}_1$: one which corresponds to the communication in the first application between the first and the second stage, and one in the second application between the third and the fourth stage. For the computation of the latency with the PATH model, it is necessary to decide in which order these communications occur. If we start with the communication in the first application, the latency is computed as follows:

$$\max\left(\frac{3}{6}+\frac{1}{1}+\frac{2+1}{3}, \frac{2}{3}+\frac{1}{1}+\frac{6+4}{6}+\frac{1}{1}+\frac{1}{1}+\frac{2}{3}\right) = 6 .$$

There is one time unit of idle time in the computation of the latency of the second application, which corresponds to the communication from $\mathcal{P}_2$ to $\mathcal{P}_1$ in the first application. The latency can be reduced to 5 if we change the order of communications. Actually, for general mappings, even if the mapping is fixed, it is NP-hard to decide in which order communications should be executed in order to minimize the latency with the PATH model [1].

Because of this observation, we consider the WAVEFRONT model when dealing with general mappings. This model was introduced by Hary and Özgüner in [55], and it is widely used in real-time systems. Note that the WAVEFRONT model assumes a full overlap of communications and computations. In the example, the latency is still dictated by the second application: this application needs 5 periods to execute a whole data set. The WAVEFRONT latency is therefore $5 \times \frac{7}{3} \approx 11.66$.

2.4 Framework

We start with a formal description of the applicative framework (Section 2.4.1) and the target execution platform (Section 2.4.2). Next in Section 2.4.3, we introduce and motivate the mapping strategies. We are then ready to formally describe the performance objective criteria (period and latency) in Section 2.4.4, and then to finally discuss the energy model in Section 2.4.5.

2.4.1 Applicative framework

We consider A independent application workflows ($A \geq 1$) to be executed concurrently; each application operates on a collection of data sets that are executed in a pipelined fashion. For $1 \leq a \leq A$, let n_a be the number of stages of application a, and $N = \sum_{a=1}^{A} n_a$ be the total number of stages. For $1 \leq k \leq n_a$, w_a^k is the computation requirement of $\mathcal{S}_a^k$, the kth stage of application a. For $1 \leq k < n_a$, δ_a^k is the size of the output data of $\mathcal{S}_a^k$.

2.4.2 Target platform

The target platform is composed of p processors, which are fully interconnected; there is a bidirectional link $\mathcal{P}_u \leftrightarrow \mathcal{P}_v$ between any processor pair $\mathcal{P}_u$ and $\mathcal{P}_v$, of bandwidth $b_{u,v}$.

We use a linear cost model for communications; it takes $X/b_{u,v}$ time units to send (resp. receive) a message of size X to (resp. from) $\mathcal{P}_v$. With the mapping rules that we enforce (see Section 2.4.3 below), it turns out that a processor never has to perform two concurrent ingoing nor outgoing communications: at any time-step, a processor is involved in at most one send, one computation and one receive. However, these three operations can either be parallel (as in the example of Section 2.3) or serialized. With parallel

operations, we have the *overlap* model that corresponds to multi-threaded communication libraries such as MPICH2 [67]. With sequential operations, we have the *no-overlap* model that is well-suited to single-threaded programs.

We assume that processors are multi-modal: every processor $\mathcal{P}_u$ is associated with a set of speeds $\mathsf{S_u} = \{s_{u,1}, \ldots, s_{u,m_u}\}$. During the mapping process, we need to choose one speed in $\mathsf{S_u}$ for each processor $\mathcal{P}_u$ that is enrolled, and this speed is fixed during the whole execution.

Then we classify particular cases which are important, both from a theoretical and practical perspective. *Fully homogeneous* platforms, also called *speed homogeneous*, have identical processors (all processors have a common speed set: $\mathsf{S_u} = \mathsf{S}$) and homogeneous communication devices ($b_{u,v} = b$ for all link bandwidths). They represent typical parallel machines. *Communication homogeneous* platforms, also called *speed heterogeneous*, are still interconnected with homogeneous communication devices, but they may have processors with different speed sets ($\mathsf{S_u} \neq \mathsf{S_v}$). They correspond to networks of workstations with plain TCP/IP interconnects or other LANs. *Fully heterogeneous platforms* are the most general, fully heterogeneous architectures. Hierarchical platforms made up with several clusters interconnected by slower backbone links can be modeled this way.

2.4.3 Mapping strategies and scheduling

We consider three mapping strategies. *One-to-one* mappings obey the simplest rule: each application stage is allocated to a distinct processor. While easier to optimize and implement, this rule may be unduly restrictive, and is likely to pay high communication costs. Obviously, it also requires that $p \geq N$, thereby limiting its applicability to larger platforms (or fewer and smaller applications). A natural extension is to search for *interval* mappings, where each participating processor is assigned an interval of consecutive stages. Intuitively, assigning several consecutive stages to the same processors will increase their computational load, but may well dramatically decrease communication requirements. Interval mappings have been widely used in the literature, see [110, 111, 17, 122, 123] among others. We point out that both one-to-one and interval mappings forbid any processor sharing, or re-use, across applications. These mappings are relevant in practice, for instance if we envision a computer center where applications, or jobs, cannot share resources because of security rules or of batch-assignment procedures. The goal of the platform manager is to secure an efficient (albeit concurrent) execution for each application (performance-related criteria) while minimizing the energy consumption of the whole platform.

We also introduce *general* mappings that allow any processor to execute any number of stages, consecutive or not, taken from one or several applications. Such mappings are likely to lead to a better resource utilization throughout the platform.

2.4.4 Performance optimization criteria

We are now ready to formally define the *period* and the *latency* of the applications. We start with one-to-one and interval mappings with no processor sharing, and then we discuss the impact of processor sharing on the metrics.

Without processor sharing

For one-to-one and interval mappings, since there is no processor sharing, we can focus on a single application.

Formally, an interval mapping is a partition of the set of stages $\mathcal{S}^1$ to $\mathcal{S}^n$ into m intervals $I_j = [d_j, e_j]$ such that $d_j \leq e_j$ for $1 \leq j \leq m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m - 1$ and $e_m = n$. Then,

the function $\mathsf{al} : [1, n] \mapsto [1, p]$ associates a processor number to each stage number. In a one-to-one mapping, this function is a one-to-one assignment. In an interval mapping, for $1 \leq j \leq m$, the whole interval I_j is mapped onto the same processor $\mathcal{P}_{\mathsf{al}(d_j)}$, i.e., for $d_j \leq i \leq e_j$, $\mathsf{al}(i) = \mathsf{al}(d_j)$. Also, two intervals (from the same application or from two different applications) cannot be mapped onto the same processor, i.e., for $1 \leq j, j' \leq m$, $j \neq j'$, $\mathsf{al}(d_j) \neq \mathsf{al}(d_{j'})$.

The period of this single application is expressed in the overlap model as:

$$T^{(overlap)} = \max_{j \in \{1,\dots,m\}} \left(\max \left(\frac{\delta^{d_j - 1}}{b_{\mathsf{al}(d_j-1),\mathsf{al}(d_j)}}, \frac{\sum_{i=d_j}^{e_j} w^i}{s_{\mathsf{al}(d_j)}}, \frac{\delta^{e_j}}{b_{\mathsf{al}(d_j),\mathsf{al}(e_j+1)}} \right) \right), \tag{2.5}$$

with $\delta^{d_1-1} = \delta^{e_m} = 0$ for the boundaries.

The maximum in the previous expression is replaced by a sum when considering the no-overlap model, since all operations are serialized. The period is then:

$$T^{(no-overlap)} = \max_{j \in \{1,\dots,m\}} \left(\frac{\delta^{d_j - 1}}{b_{\mathsf{al}(d_j-1),\mathsf{al}(d_j)}} + \frac{\sum_{i=d_j}^{e_j} w^i}{s_{\mathsf{al}(d_j)}} + \frac{\delta^{e_j}}{b_{\mathsf{al}(d_j),\mathsf{al}(e_j+1)}} \right). \tag{2.6}$$

The latency is the time to process a single data entirely, so it is identical in both communication models, and computed with the PATH model:

$$L = \sum_{j=1}^{m} \left(\sum_{i=d_j}^{e_j} \frac{w^i}{s_{\mathsf{al}(d_j)}} + \frac{\delta^{e_j}}{b_{\mathsf{al}(d_j),\mathsf{al}(e_j+1)}} \right) , \tag{2.7}$$

with $\delta^{e_m} = 0$ for the boundary.

Again, the simplicity of Equations (2.5), (2.6) and (2.7) is a very useful property of interval mappings, and greatly simplifies the solution of multi-criteria problems.

These are the period and latency of one single application, and we need to define a global period and latency function to be optimized. The simplest approach is to minimize $X = \max_{a \in \{1,\dots,A\}}(X_a)$, where X_a is the period or latency of application a, for $a \in \{1, \dots, A\}$, as in the example of Section 2.3. However, the concurrent applications can be of completely different nature and/or economic value, so that their periods or latencies are not always comparable. Therefore we aim at minimizing

$$X = \max_{a \in \{1,\dots,A\}} W_a \times X_a, \tag{2.8}$$

where $W_a > 0$ is a weight associated to each application and X_a is the period or latency of application a, for $a \in \{1, \dots, A\}$. W_a can be 1 (we retrieve a simple maximum) or a priority ratio (fixed by the platform manager and/or paid by the user). We can also let $W_a = 1/X_a^*$, where X_a^* is the objective function computed when the application is executed alone on the platform; in this case $W_a \times X_a$ represents the slowdown factor of application a, and X corresponds to the maximum stretch [15].

With resource sharing

If we keep the classical latency definition (PATH model) and consider general mappings, it leads to intricate scheduling problems for period/latency bi-criteria problems. Basically, even when the mapping is given, scheduling the execution is a problem of combinatorial nature (it is NP-complete, see [1]). With general mappings, a processor typically has several incoming and/or outgoing communications, and it

is difficult to orchestrate these operations so as to minimize conflicting objectives such as period and latency. This holds true both for the overlap and no-overlap models.

Therefore, when considering resource sharing, we focus in this chapter on the problem in which bounds on period and latency are fixed by the application designer, and we relax the definition of the latency using the approach of Hary and Özgüner [55], that we call the WAVEFRONT model. Instead of computing the longest path, we approximate the latency L as $L = (2m - 1)T$, where T is the period, i.e., the rate at which data sets enter the system, and m is the number of intervals of consecutive stages mapped onto a same processor in the mapping. A processor change occurs each time when a stage and its successor are not mapped onto the same processor, i.e., $m - 1$ times. The intuition is that the whole application is executed synchronously, and each data set progresses concurrently within a period. With m successive computations and $m-1$ processor changes (i.e., communications), each data set traverses the platform within $2m - 1$ periods.

The mapping is an allocation function, which associates a processor number to each stage number, as well as a speed at which each processor is running. For general mappings with processor reuse, we must carefully decide how the speed of each processor is shared among all stages it is assigned to. Similarly, a communication link or processor network card may be involved in several communications, which implies to sharing bandwidths and card capacities, too. Hence the question is the following: given the mapping, and a threshold period T_a and latency L_a for each application $a \in \{1, \ldots, A\}$, is it possible to determine which fraction of computing and communicating resources to assign to each operation so that all period and latency thresholds are met?

Since we consider the WAVEFRONT latency model, one period is accounted for each computation of an interval of stages and for each inter-processor communication. We observe that given the mapping, we know m_a, the number of intervals ($m_a - 1$ processor changes), for each application a. We can thus check immediately whether the bounds on the latency are respected, i.e., $(2m_a - 1)T_a \leq L_a$ for $a \in \{1, \ldots, A\}$.

Now for the periods, the key idea is to distribute platform resources parsimoniously, and to allocate only the needed CPU fraction to each computation, and the needed bandwidth fraction to each communication, so that the period constraint is fulfilled. The mapping is valid if neither processor speeds, nor link bandwidths, nor network card capacities are exceeded. First, we merge consecutive stages $[\mathcal{S}_a^i, \ldots, \mathcal{S}_a^j]$ of application a mapped onto a same processor as one single coalesced stage $\hat{\mathcal{S}_a^k}$, with computing cost $\hat{w_a^k} = \sum_{k'=i}^{j} w_a^{k'}$, and output communication cost $\hat{\delta_a^k} = \delta_a^j$. The transformed application now has exactly m_a stages. In the following, stage $\hat{\mathcal{S}_a^k}$ corresponds to the k-th stage of the transformed application a, for $1 \leq k \leq m_a$.

As for computations, consider a processor $\mathcal{P}_u$ and an application a. We define $\mathcal{K}_a^u$ such that $k \in \mathcal{K}_a^u$ if and only if $\hat{\mathcal{S}_a^k}$ is processed by processor $\mathcal{P}_u$; $\mathcal{K}_a^u$ is the set of stages of (transformed) application a processed by $\mathcal{P}_u$. Then, for all a and u, and for each $k \in \mathcal{K}_a^u$, we allocate the speed fraction $s_{a,u}^k = \hat{w_a^k}/T_a$ for $\mathcal{P}_u$ to execute $\hat{\mathcal{S}_a^k}$.

Similarly for communications, we define $\mathcal{K}_a^{u,v}$ such that $k \in \mathcal{K}_a^{u,v}$ if and only if $\hat{\mathcal{S}_a^k}$ is processed by $\mathcal{P}_u$ and $\hat{\mathcal{S}_a^{k+1}}$ is processed by $\mathcal{P}_v$, i.e., there is a communication to pay between $\mathcal{P}_u$ and $\mathcal{P}_v$. Note that $u \neq v$, otherwise stages $\hat{\mathcal{S}_a^k}$ and $\hat{\mathcal{S}_a^{k+1}}$ would have been merged as a single stage. Formally, $k \in \mathcal{K}_a^{u,v} \Leftrightarrow k \in \mathcal{K}_a^u$ and $k+1 \in \mathcal{K}_a^v$. Then we allocate the bandwidth fraction $b_{a,u,v}^k = \hat{\delta_a^k}/T_a$ to the communication.

The period of each application can be respected if and only if all the following inequalities are satisfied. There might be some spare speed and bandwidth if these are strict inequalities, and resources

are fully utilized in case of equalities:

- $\forall 1 \leq u \leq p,\ \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^u} s_{a,u}^k \leq s_u,\ \ \sum_{v=1}^{p} \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^{u,v}} b_{a,u,v}^k \leq B_u^{out},$ $\sum_{v=1}^{p} \sum_{a=1}^{A} \sum_{k \in \mathcal{K}_a^{v,u}} b_{a,v,u}^k \leq B_u^{in};$
- $\forall 1 \leq u, v \leq p,\ u \neq v,\ \sum_{a=1}^{A} \left(\sum_{k \in \mathcal{K}_a^{u,v}} b_{a,u,v}^k + \sum_{k \in \mathcal{K}_a^{v,u}} b_{a,v,u}^k \right) \leq b_{u,v}.$

Note that we can consider mappings without reuse with this latency model. In this case, if we transform each application a as explained above, the allocation function of stages $\hat{\mathcal{S}}_a^k$ (for $1 \leq a \leq A$ and $1 \leq k \leq m_a$) is a one-to-one function: each coalesced stage is allocated onto a distinct processor. It becomes then much easier to check the validity of the mapping, since each processor is only handling one single stage, receiving input data from one single other processor, and sending output data to one single other processor.

2.4.5 Energy model

The energy consumption of the platform is defined as the sum of the energy $E(u, \ell)$ consumed by each processor $\mathcal{P}_u$ enrolled in the mapping in mode ℓ. We assume that $E(u, \ell)$ consists of a static part and of a dynamic part. The static part $E_{stat}(u)$ is the static cost for a processor to be in service, and does not depend on the speed $s_{u,\ell}$ at which the processor is running. However, the static energy is consumed only in mode $\ell \neq 0$ (otherwise, the processor is inactive, and not enrolled in the mapping). In Chapter 1, we did not consider this static energy: we assumed that the processors could not be turned off, hence the dynamic energy was the objective function. This dynamic part $E_{dyn}(u, \ell)$ is here of the form $E_{dyn}(u, \ell) = s_{u,\ell}^{\alpha}$, where $\alpha > 1$ is an arbitrary rational number. It is often assumed that $\alpha = 3$ [62, 40, 87, 95], as we did in Chapter 1 and in the example of Section 2.3, but all our results hold for any value of α. Finally, for $\ell \neq 0$, we have $E(u, \ell) = E_{stat}(u) + E_{dyn}(u, \ell)$, while $E(u, 0) = 0$.

The energy $E(u, \ell)$ is an energy consumed per time unit, so we could also speak of dissipated power. Note that it is mandatory to minimize energy consumption per time unit, because the execution of streaming applications with arbitrarily many data sets may last for an unbounded amount of time. Hence we always consider a combination of energy and period objective criteria, because the latency by its own takes only one single data set into account, and does not reflect a pipelined execution.

2.5 Complexity results with the PATH model

In this section, we consider the PATH model for the computation of the latency, and therefore we restrict the study to one-to-one and interval mappings with no resource sharing. General mappings with resource sharing are investigated in Section 2.6.

In the following, **proc-hom** denotes identical speed processors while **proc-het** represents heterogeneous processors; **com-hom** means identical communication links, while they differ for **com-het**. We also report results for the case **special-app**, which corresponds to applications whose stages are all identical (all w_a^k are equal), and no communication cost is paid (all δ_a^k are equal to 0).

We start with the mono-criterion problems of period or latency minimization in Sections 2.5.1 and 2.5.2. In these cases, we do not consider energy minimization issues, and therefore we can systematically run processors at their highest speed, and thus use classical results established in a context with no energy. Then we investigate the following multi-criteria problems: period/latency (Section 2.5.3), period/energy (Section 2.5.4) and period/latency/energy (Section 2.5.5). We discard the latency/energy combination since, as discussed above, the energy model holds only in combination with the period criterion.

When dealing with multiple criteria, our approach is to minimize one of them, given a threshold on the others. Actually, fixing the period or the latency means fixing a threshold on the period or latency of each application, thus providing a table of period or latency values. Equivalently, we minimize the value of Equation (2.8) with suitable coefficients. For the energy, only a bound on the global energy consumption is required. Note that all results apply to both the overlap and no-overlap models, and to all objective functions introduced in Section 2.4.4: more precisely, polynomial problems remain polynomial for arbitrary weights W_a in Equation (2.8), while NP-complete problems are already difficult with $W_a = 1$. All complexity results are summarized in Section 2.5.6.

2.5.1 Period minimization

We show that a greedy assignment solves the problem of finding a one-to-one mapping on communication homogeneous platforms, but the problem turns NP-complete with heterogeneous links between the processors. For interval mappings, we use an existing algorithm which finds the minimum period in a single application to build a new polynomial time algorithm that minimizes the global period of many applications on fully homogeneous platforms, giving the right number of processors to each application. The problem is NP-complete with heterogeneous processors, even for the case **special-app**.

One-to-one mappings

Theorem 2.1. *On communication homogeneous platforms, a one-to-one mapping that minimizes the period can be determined in polynomial time.*

Proof. The following proof is an adaptation of the algorithm described in [17], which finds the minimum period under the same hypothesis but for a single application. The main idea remains the same, since on communication homogeneous platforms the application that the stage belongs to does not matter for a one-to-one mapping.

The optimal period belongs to the set:

$$\mathcal{T} = \left\{ W_a \times \max\left(\frac{\delta_a^{k-1}}{b}, \frac{w_a^k}{s_u}, \frac{\delta_a^k}{b} \right) \right\}_{1 \leq a \leq A, 1 \leq k \leq n_a, 1 \leq u \leq p},$$

because it is equal to the product of W_a by the cycle-time of some processor $\mathcal{P}_u$, running in its fastest mode s_u, and executing one of the N stages, S_a^k. First we compute the set $\mathcal{T}$ and we sort its elements into an array $\mathcal{T}_A$. Then, we perform a binary search on the array $\mathcal{T}_A$ to find the optimal period, testing at each step whether the current element T is a feasible value. To do so, we use the greedy assignment procedure of Algorithm 1. Initially, the current element T is the median of $\mathcal{T}_A$. If the greedy assignment procedure returns "failure", we increase the period by jumping to the median of the elements of $\mathcal{T}_A$ which are larger than T, and if it returns "success", we jump to the median of the elements of $\mathcal{T}_A$ which are smaller than T. The algorithm terminates in $\lceil \log \mathcal{T} \rceil$ iterations.

Note that $|\mathcal{T}| \leq N \times p$ (N stages and p processors), hence the total computation time is $O((N \times p + cost_{GA}) \log(N \times p))$, where $cost_{GA}$ is the cost of the greedy assignment procedure.

We now describe the greedy assignment algorithm for a prescribed value T of the achievable period. Recall that there are N stages to map onto $p \geq N$ processors in a one-to-one fashion. Also, we target communication homogeneous platforms with different-speed processors ($s_u \neq s_v$), with different-capacity links between the applications, but with links of same capacities within an application. First we retain only the fastest N processors, which we rename $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N$ such that $s_1 \leq s_2 \leq \cdots \leq s_N$. Then we consider the processors in the order $\mathcal{P}_1$ to $\mathcal{P}_N$, i.e., from the slowest to the fastest, and greedily assign them any free (not already assigned) task that they can process within the period.

Algorithm 1: Greedy-Assignment(T)

Work with fastest N processors, numbered $\mathcal{P}_1$ to $\mathcal{P}_N$, where $s_1 \leq s_2 \leq \cdots \leq s_N$
Mark all stages $\mathcal{S}_1$ to $\mathcal{S}_N$ as free
for $u = 1$ to N **do**
  Pick up any free stage $\mathcal{S}_a^k$ such that:
    $W_a \times \max(\frac{\delta_a^{k-1}}{b_a}, \frac{w_a^k}{s_u}, \frac{\delta_a^k}{b_a}) \leq T$
  Assign $\mathcal{S}_a^k$ to $\mathcal{P}_u$
  Mark $\mathcal{S}_a^k$ as already assigned
  if no stage found **then**
    return "failure"
  end if
end for
return "success"

The proof that the greedy procedure returns a solution if and only if there exists a solution of period T is done by a simple exchange argument. Indeed, consider a valid one-to-one assignment of period T, denoted $\mathcal{A}$, and assume that it has assigned stage $\mathcal{S}_{a_1}^{k_1}$ to $\mathcal{P}_1$. Note first that the greedy procedure will indeed find a stage to assign to $\mathcal{P}_1$ and cannot fail, since $\mathcal{S}_{a_1}^{k_1}$ can be chosen. If the choice of the greedy procedure is actually $\mathcal{S}_{a_1}^{k_1}$, we proceed by induction with $\mathcal{P}_2$. If the greedy procedure has selected another stage $\mathcal{S}_{a_2}^{k_2}$ for $\mathcal{P}_1$, we find which processor, say $\mathcal{P}_u$, has been assigned this stage in the valid assignment $\mathcal{A}$. Then we exchange the assignments of $\mathcal{P}_1$ and $\mathcal{P}_u$ in $\mathcal{A}$. As $\mathcal{P}_u$ is faster than $\mathcal{P}_1$, which could process $\mathcal{S}_{a_1}^{k_1}$ in time in the assignment $\mathcal{A}$, $\mathcal{P}_u$ can process $\mathcal{S}_{a_1}^{k_1}$ in time too.

As $\mathcal{S}_{a_2}^{k_2}$ has been mapped on $\mathcal{P}_1$ by the greedy procedure, $\mathcal{P}_1$ can process $\mathcal{S}_{a_2}^{k_2}$ in time. So the exchange is valid, we can consider the new assignment which is valid and which did the same assignment on $\mathcal{P}_1$ than the greedy procedure. The proof proceeds by induction with $\mathcal{P}_2$ as before.

The complexity of the greedy assignment procedure is $cost_{GA} = O(N^2)$, because of the two loops over processors and stages. Altogether, since $N \leq p$, the cost of Algorithm 1 can be neglected, and the complexity of the whole algorithm is $O((N \times p)\log(N \times p))$, which is indeed polynomial in the problem size.

In addition, note that this algorithm works with the no-overlap communication model, by replacing $W_a \times \max(\frac{\delta_a^{k-1}}{b_a}, \frac{w_a^k}{s_u}, \frac{\delta_a^k}{b_a}) \leq T$ by $W_a \times (\frac{\delta_a^{k-1}}{b_a} + \frac{w_a^k}{s_u} + \frac{\delta_a^k}{b_a}) \leq T$. ■

Theorem 2.2. *On fully heterogeneous platforms, the problem of finding a one-to-one mapping that minimizes the period is NP-complete.*

Proof. As the problem was already NP-complete with one single application [17], it remains NP-complete with concurrent applications. ■

Interval mappings

Theorem 2.3. *On fully homogeneous platforms, an interval mapping that minimizes the period can be determined in polynomial time.*

Proof. A polynomial algorithm has already been found to exhibit the minimal period with one application, under a communication model without overlap [17], and it can easily be extended to the overlap model, so the following proof is valid for both models. We exhibit an algorithm (see Algorithm 2) which finds an optimal interval mapping for concurrent applications, thanks to the previous polynomial algorithm for a single application, and we show its validity.

Algorithm 2:

Assign all stages of each application to one processor
Compute the period of all applications
for $a = (p - A)$ to p **do**
  Find an application a' such that $W_{a'} \times T_{a'}$ is maximum
  Add one processor to this application
  Compute the new period $T_{a'}$ of this application
end for

First, here are some notations:

– $(k^u_{a,i})$ is a A-tuple which represents the processor distribution among the applications at step i of Algorithm 2.
– $(k^o_{a,i})$ is an optimal processor distribution with i processors.
– $T_a(n)$ is the period of the application numbered a, where n is the number of processors the application a is assigned to.
– $T(d) = \max_{a\in\{1,\dots,A\}} W_a \times T_a(d_a)$, where d is a A-tuple.

Let us prove now the optimality of Algorithm 2.

– $(k^u_{a,A})$ is the best distribution with A processors, because it is the only one.
– Let us assume that $(k^u_{a,i})$ is optimal with i processors. We want $(k^u_{a,i+1})$ to be an optimal distribution with $i + 1$ processors.
  – Either: $\exists a, k^o_{a,i+1} < k^u_{a,i}$
    In this case, by construction,

$$\exists i' < i, T((k^u_{a,i'})) = W_a \times T_a(k^u_{a,i'}) = W_a \times T_a(k^o_{a,i+1})$$

    Now, because every T_a and $x \mapsto W_a \times x$ are non-decreasing, $T((k^u_{a,i+1})) \le T((k^u_{a,i})) \le T((k^u_{a,i'}))$, and by definition $W_a \times T_a(k^o_{a,i+1}) \le T((k^o_{a,i+1}))$.
    Finally, $T((k^u_{a,i+1})) \le T((k^o_{a,i+1}))$.
  – Or: $\exists! a, k^o_{a,i+1} = k^u_{a,i} + 1$
    – either: $k^u_{a,i+1} = k^u_{a,i} + 1$ and we are done,
    – or: $\exists a' \ne a, k^u_{a',i+1} = k^u_{a',i} + 1$
      In this case, by construction,
      $T((k^u_{a,i})) = f_{a'}(T_{a'}(k^u_{a',i})) = f_{a'}(T_{a'}(k^o_{a',i+1}))$ because $k^u_{a',i} = k^o_{a',i+1}$. Thus $T((k^u_{a,i})) \le T((k^o_{a,i+1}))$. Finally, $T((k^u_{a,i+1})) \le T((k^o_{a,i+1}))$.

  Overall we have shown that $(k^u_{a,i+1})$ was as good as $(k^o_{a,i+1})$.
– By induction, the algorithm finds an optimal solution to map A applications onto p processors.

The complexity of computing the period of application a with $q \le p$ processors, keeping the intermediate result with $q - 1$ processors, is bounded by $O((n_a)^3 q)$ [17]. Let $n_{max} = \max_{a\in\{1,\dots,A\}} n_a$. Since we perform at most p steps in the algorithm, and $q \le p$, the complexity of Algorithm 2 is bounded by $O(n^3_{max} p^2)$, which is indeed polynomial in the problem size. ■

Theorem 2.4. *On communication homogeneous platforms, the problem of finding an interval mapping that minimizes the period is NP-complete.*

Proof. As the problem was already NP-complete with one single application [17], it remains NP-complete with concurrent applications. ■

The case **special-app** is more interesting, because a polynomial algorithm exists to find an interval mapping which minimizes the period of one single application [18]; however, the problem becomes NP-complete with several applications.

Theorem 2.5. *With more than one application, heterogeneous processors, homogeneous pipelines without communication, finding an interval mapping which minimizes respectively* $\max_{a\in\{1,\dots,A\}} T_a$*, or* $\max_{a\in\{1,\dots,A\}} W_a \times T_a$*, or* $\max_{a\in\{1,\dots,A\}} T_a/T_a^*$*, is a NP-complete problem (in the strong sense).*

Proof. First we focus on the first problem, i.e., minimizing $\max_{a\in\{1,\dots,A\}} T_a$.

We consider the associated decision problem: given a period T, is there a mapping of period less than T? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time that it is valid by computing its period.

To establish the completeness, we use a reduction from 3-PARTITION [44]. We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3m$ positive integers $a_1, a_2, \dots, a_{3m}$ such that for all $i \in \{1, \dots, 3m\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{m} a_i = mB$, does there exist a partition $I_1, \dots, I_m$ of $\{1, \dots, 3m\}$ such that for all $j \in \{1, \dots, m\}$, $|I_j| = 3$ and $\sum_{i\in I_j} a_i = B$?

As 3-PARTITION is NP-complete in the strong sense, we can encode the $3m$ numbers in unary, and assume that the size of $\mathcal{I}_1$ is $O(mB)$.

We build an instance $\mathcal{I}_2$ of our problem with m identical applications such that each application is composed of B stages, with $w = 1$, and $p = 3m$ processors with speeds a_j for each $j \in \{1, \dots, 3m\}$. We ask whether it is possible to realize a period of 1. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$ (coded in unary). We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. Let $I_j = \{a'_{1,j}, a'_{2,j}, a'_{3,j}\}$, for $j \in \{1, \dots, m\}$. For each $j \in \{1, \dots, m\}$, we assign the $a'_{1,j}$ first consecutive stages of the application j to the processor of speed $a'_{1,j}$, the $a'_{2,j}$ next stages to the processor of speed $a'_{2,j}$, and the $a'_{3,j}$ remaining stages to the processor of speed $a'_{3,j}$. As the period of every processor is clearly equal to 1, the period is 1.

Suppose now that $\mathcal{I}_2$ has a solution. As the sum of all computation times is equal to the sum of all processor speeds, and a processor cannot be assigned stages of two different applications, for each application, the sum of its computation times is equal to the sum of the speed of processors which are assigned a stage of this application. Now, for all $i \in \{1, \dots, 3m\}$, $B/4 < a_i < B/2$, so there are exactly three processors involved in the processing of each application. We can derive easily a solution to $\mathcal{I}_1$ (set I_j corresponding to processors of application j).

As there is no communication, this proof is valid for both communication models.

For the second problem, we follow the previous proof, but we assume now that, for each $a \in \{1, \dots, A\}$, for $k \in \{1, \dots, m\}$, $w_a^k = 1/W_a$. Then we scale each application: each w_a^k is multiplied by W_a so that the new period T'_a of the application a will be $W_a T_a$. We are now in the case of the previous proof.

Finally, for the third problem, we build the same instance as the one of the first proof. As the pipeline applications are all similar, the period of those applications when they are alone on the platform are all the same. We finally just have to minimize $\max_{a\in\{1,\dots,A\}} T_a$. ■

2.5.2 Latency minimization

We show that finding a one-to-one mapping which minimizes the latency is NP-complete as soon as the processors do not have the same speed thanks to a reduction from 3-PARTITION. However we write a greedy algorithm that finds the optimal interval mapping on communication homogeneous platforms. The problem is still NP-complete on fully heterogeneous platforms for interval mappings.

Note that latency expression does not depend on the communication model, thus the results of this section are valid for the overlap and no-overlap models.

One-to-one mappings

Theorem 2.6. *The problem of finding the one-to-one mapping which minimizes the latency on fully homogeneous platforms is polynomial.*

Proof. As all mappings are equivalent, the theorem is true. ■

The case **special-app** is more interesting, because a polynomial algorithm exists to find a one-to-one mapping which minimizes the latency of one single application [19]; however, the problem becomes NP-complete with several concurrent applications.

Theorem 2.7. *With several applications, heterogeneous processors, homogeneous pipelines without communication, the problem of finding the optimal one-to-one mapping which minimizes respectively* $\max_{a\in\{1,\dots,A\}} L_a$, $\max_{a\in\{1,\dots,A\}} W_a \times L_a$, *or* $\max_{a\in\{1,\dots,A\}} L_a/L_a^*$, *are NP-complete (in the strong sense).*

Proof. First we focus on the first problem, i.e., minimizing $\max_{a\in\{1,\dots,A\}} L_a$.

We consider the associated decision problem: given a latency L, is there a mapping of latency less than L? The problem is obviously in NP: given a latency and a mapping, it is easy to check in polynomial time that it is valid by computing its latency.

To establish the completeness, we use a reduction from 3-PARTITION. We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3m$ positive integers $a_1, a_2, \dots, a_{3m}$ such that for all $i \in \{1,\dots,3m\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{m} a_i = mB$, does there exists a partition $I_1,\dots,I_m$ of $\{1,\dots,3m\}$ such that for all $j \in \{1,\dots,m\}$, $|I_j| = 3$ and $\sum_{i\in I_j} a_i = B$?

We build an instance $\mathcal{I}_2$ of our problem with m identical applications, each one composed of 3 stages with $w = 1$, and $p = 3m$ processors with speeds $1/a_j$ for $j \in \{1,\dots,3m\}$. We ask whether it is possible to realize a global latency of B. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. Let, for each $j \in \{1,\dots,m\}$, $I_j = \{a'_{1,j}, a'_{2,j}, a'_{3,j}\}$. For each $j \in \{1,\dots,m\}$, for $i \in \{1,2,3\}$ we assign the ith stage of the application j to the processor whose speed is equal to $1/a'_{i,j}$. The global latency is clearly B.

Suppose now that $\mathcal{I}_2$ has a solution. There exists a partition $I_1,\dots,I_m$ of $\{1,\dots,3m\}$ such that for all $j \in \{1,\dots,m\}$, $|I_j| = 3$ and $\sum_{i\in I_j} a_i \le B$. Since $\sum_{i=1}^{m} a_i = mB$, we have, $\forall j \in \{1,\dots,m\}$, $\sum_{i\in I_j} a_i = B$. We conclude that $\mathcal{I}_1$ has a solution.

For the second problem, the proof is the same as the previous one, but we have now $w_a^1 = w_a^2 = w_a^3 = 1/W_a$.

For the third problem, the proof is similar to the first one, but we ask now whether it is possible to realize a global latency of $K \times B$, where K is the sum of the three biggest a_i. All applications

have indeed the same latency when they are alone on the platform, and this latency is K. Instead of minimizing $\max_{a\in\{1,\dots,A\}} \frac{L_a}{L_a^*}$, we minimize $\max_{a\in\{1,\dots,A\}} \frac{L_a}{K}$ so we minimize $\max_{a\in\{1,\dots,A\}} L_a$. ■

Interval mappings

Theorem 2.8. *On communication homogeneous platforms, the optimal interval mapping which minimizes the latency can be determined in polynomial time.*

Proof. First, note that with a single application, the optimal mapping is obtained by mapping the whole application onto one processor. Indeed, if two distinct processors were enrolled in the computation, mapping the entire application onto the fastest processor would reduce the computation time and remove the communication cost. Therefore, with several concurrent applications, we keep the A fastest processors and map the applications onto those processors in a one-to-one fashion. The greedy procedure written for the period minimization problem with one-to-one mapping can be reused.

The optimal latency belongs to the set:

$$\mathcal{L} = \left\{ W_a \times \left(\frac{\delta_a^0}{b} + \frac{\sum_{k=1}^{n_a} w_a^k}{s_u} + \frac{\delta_a^{n_a}}{b} \right) \right\}_{1\leq a\leq A,\ 1\leq u\leq p}.$$

Since $|\mathcal{L}| = Ap$, the complexity of the algorithm is $O((Ap + A^2)\log(Ap))$, and it can be simplified in $O(Ap\log(Ap))$. ■

Theorem 2.9. *On fully heterogeneous platforms, the problem of finding an optimal interval mapping, that minimizes the latency, is NP-complete.*

Proof. As the problem of finding the interval mapping, which minimizes the latency on fully heterogeneous platforms, was already NP-complete with one single application [19], it remains NP-complete with several concurrent applications. ■

2.5.3 Period/latency minimization

In this section again, we are not concerned with energy minimization issues, so, similarly to results of Sections 2.5.1 and 2.5.2, all processors can be run systematically at their highest speed. Therefore, on fully homogeneous platforms, all one-to-one mappings are identical, and it is straightforward to minimize the latency for a given period, or the converse.

However, for interval mappings, we must decide where to split applications into intervals, and we provide a dynamic programming algorithm which solves both variants of the problem with a single application. When considering multiple applications, we need to run the dynamic programming algorithm once per application with the corresponding period (resp. latency) threshold, and the minimum latency (resp. period) that can then be achieved is the maximum over all applications.

Theorem 2.10. *With one application, on fully homogeneous platforms, the optimal interval mapping which minimizes the latency for a bounded period, or the period for a bounded latency, can be determined in polynomial time.*

Proof. We denote by n the number of stages, s the speed of every processor and b their bandwidth.

We exhibit a dynamic programming algorithm which computes the optimal mapping that minimizes the latency for a given period. We compute recursively the values of $(L,T)(i,q)$, which are the optimal latency and period that can be achieved by any interval-based mapping of stages $\mathcal{S}^1$ to $\mathcal{S}^i$ using exactly q processors. The recurrence relation can be expressed as:

$$(L,T)(i,q) = \min_{1\leq j<i} \left\{ \begin{array}{l} \Big(L(j,q-1) + \frac{\sum_{k=j+1}^{i} w^k}{s} + \frac{\delta^i}{b}, \\ \max \quad \Big(T(j,q-1), \max(\frac{\delta^j}{b}, \frac{\sum_{k=j+1}^{i} w^k}{s}, \frac{\delta^i}{b})\Big)\Big) \end{array} \right\}.$$

This relation holds for all $i > 1$ and $q > 1$. The function "min" keeps the brace such that the period is not greater than the given period and the latency is minimum. If such a brace does not exist, it returns $(+\infty,+\infty)$.

The initialization relations are:

– If there is only one processor, we map the whole interval onto this processor. For each $i \in \{1,\dots,n\}$:

$$(L,T)(i,1) = \left(\frac{\delta^0}{b} + \frac{\sum_{k=1}^{i} w^k}{s} + \frac{\delta^i}{b}, \max\left(\frac{\delta^0}{b}, \frac{\sum_{k=1}^{i} w^k}{s}, \frac{\delta^i}{b}\right)\right)$$

– If $q > 1$ (too many processors for one stage):

$$(L,T)(1,q) = (+\infty,+\infty)$$

Finally we aim at computing:

$$\min_{q\in\{1,\dots,p\}} (L,T)(n,q) \; .$$

This dynamic programming algorithm solves the problem of finding a mapping, which minimizes the latency for a given period, with a complexity in $O(n^2 p)$.

For the converse problem of finding a mapping which minimizes the period for a given latency, we use a binary search. The minimum period belongs to the set:

$$\mathcal{T} = \left\{\frac{\sum_{k=i}^{j} w^k}{s}\right\}_{1\leq i\leq j\leq n} \bigcup \left\{\frac{\delta^i}{b}\right\}_{0\leq i\leq n}$$

Moreover, if a mapping realizes a period T and a latency L, then it realizes a period $T_2 > T$ and a latency $L_2 = L$. We conclude that the algorithm which minimizes the latency for a given period T_{lim} will find a bigger latency than the one which minimizes the latency for a given period $T_{lim}^2 > T_{lim}$. We can thus minimize the period for a given latency thanks to a binary search on the period and some calls to the previous algorithm, which minimizes the latency for a given period.

Since $|\mathcal{T}| = \frac{n(n+1)}{2} + n$, the complexity of this problem is $O((n^2+n^2p)\log(n))$, i.e. $O(n^2 p\log(n))$.

The proof of this theorem under the no-overlap communication model is very similar: all we have to do is to replace $\max(\frac{\delta^j}{b}, \frac{\sum_{k=j+1}^{i} w^k}{s}, \frac{\delta^i}{b})$ by $\frac{\delta^j}{b} + \frac{\sum_{k=j+1}^{i} w^k}{s} + \frac{\delta^i}{b}$ in the recurrence relation, $\max\left(\frac{\delta^0}{b}, \frac{\sum_{k=1}^{i} w^k}{s}, \frac{\delta^i}{b}\right)$ by $\frac{\delta^0}{b} + \frac{\sum_{k=1}^{i} w^k}{s} + \frac{\delta^i}{b}$ in the first initialization relation, and the previous $\mathcal{T}$ by

$$\mathcal{T} = \left\{\frac{\delta^{i-1}}{b} + \frac{\sum_{k=i}^{j} w^k}{s} + \frac{\delta^j}{b}\right\}_{1\leq i\leq j\leq n} .$$ ■

Theorem 2.11. *With several applications, on fully homogeneous platforms, the optimal interval mapping which minimizes the latency* $L = \max_{a \in \{1,\dots,A\}} W_a \times L_a$ *for a bounded period by application, or the period* $T = \max_{a \in \{1,\dots,A\}} W_a \times T_a$ *for a bounded latency by application can be determined in polynomial time.*

Proof. For several applications, we can reuse the structure of Algorithm 2, but instead of computing the period, we compute both period and latency, thanks to one of the previous algorithms for one single application (dynamic programming algorithm if we minimize the global latency for given periods, and binary search combined with dynamic programming algorithm if we minimize the global period for given latencies, see proof of Theorem 2.10). While there are some processors which are not yet allocated, we add one processor to any application which maximizes the criterion we want to minimize (if the bound on the other criterion is exceeded, the first criterion is set to $+\infty$, according to the single application algorithm).

Since there is a total of p calls to the single-application algorithms, and a total of N application stages, the complexity is in $O((Np)^2 \log(N))$ for the period minimization with a bounded latency, and in $O((Np)^2)$ for the latency minimization with a bounded period. ■

When moving to a platform with heterogeneous processors, even if the application is homogeneous with no communication (case **special-app**), the problem of finding a one-to-one or interval mapping that solves the bi-criteria period/latency problem is NP-complete. This result is a direct consequence of the NP-completeness of the mono-criterion cases, see Sections 2.5.1 and 2.5.2.

Theorem 2.12. *With heterogeneous processors and homogeneous pipelines, without communication, the problem of finding an interval or one-to-one mapping, that solves the bi-criteria period/latency problem, is NP-complete.*

Proof. The problem of minimizing the latency with a one-to-one mapping is NP-complete, so finding a one-to-one mapping that minimizes the latency for a given array of period is NP-complete too.

The problem of minimizing the period with an interval mapping is NP-complete, so finding an interval mapping that minimizes the period for a fixed latency by application is NP-complete too. ■

2.5.4 Period/energy minimization

We first provide results for one-to-one mappings, and then discuss interval mappings. For fully heterogeneous platforms, the problem is NP-hard because the period minimization problem already is NP-hard on such platforms. The interesting result is the following:

Theorem 2.13. *On communication homogeneous platforms, a one-to-one mapping which minimizes the energy consumption while enforcing a given period for each application can be determined in polynomial time.*

Proof. We build a bipartite graph $G = (U, V, E)$, and prove that the problem amounts to finding a minimum weighted matching in this graph. U is the processor set, and V the stage set. For each processor and each stage, the weight of the edge between the two vertices is set to $+\infty$ if the processor cannot execute the stage within the period, and else it is the energy consumed by the processor when it is running in the smallest mode allowing to execute the stage within the period. Finding a minimum weighted matching gives us the minimum power consumption, in polynomial time $O\left((N + p)^{\frac{3}{2}}\right)$. ■

For interval mappings, first note that the problem becomes NP-complete as soon as we consider different speed processors, because of the NP-completeness of the period minimization problem for such platforms. Thus we focus on fully homogeneous platforms.

Theorem 2.14. *On fully homogeneous platforms, an interval mapping which minimizes the energy consumption while enforcing a given period for each application can be determined in polynomial time.*

Proof. We first exhibit a dynamic programming algorithm that returns the optimal energy consumption for a single application, when using exactly k processors to compute the application. This algorithm is fixing the processor speeds so as to minimize the energy. Then, the multiple application case can be solved using another dynamic programming algorithm, which decides how many processors should be allocated to each application.

For a single application $a \in \{1, \dots, A\}$, and a processor number $q \in \{1, \dots, p\}$, we compute E_a^q, the minimum energy consumed for the application a using at most q processors. We recursively compute the value $E(i, j, k)$, which is the optimal energy consumption that can be achieved by any interval-based mapping of stages $\mathcal{S}_a^i$ to $\mathcal{S}_a^j$ using exactly k processors. The goal is to determine $E_a^q = \min_{k \in \{1,\dots,q\}} E(1, n_a, k)$. The recurrence relation can be expressed as:

$$E(i,j,k) = \min_{i \le \ell \le j-1} \left(E(i,\ell,k-1) + E(\ell+1,j,1) \right)$$

with the initialization:

- $E(i,i,r) = +\infty$ if $r > 1$
- Defining

$$\mathcal{F}_i^j = \left\{ E_{dyn}(s_\ell) + E_{stat} \mid \max\left(\frac{\delta^{i-1}}{b}, \frac{\sum_{k=i}^{j} w^k}{s_\ell}, \frac{\delta^j}{b} \right) \le T \right\}_{1 \le \ell \le m}$$

  we have:

$$E(i,j,1) = \begin{cases} \min \mathcal{F}_i^j & \text{if } \mathcal{F}_i^j \neq \varnothing \\ +\infty & \text{otherwise} \end{cases}$$

Here, m is the number of speed modes, and T is the period bound for the application a. The complexity of this dynamic programming algorithm is bounded by $O(n_a^2(p+m))$.

Note that for the no-overlap model, we simply replace $\max\left(\frac{\delta^{i-1}}{b}, \frac{\sum_{k=i}^{j} w^k}{s_\ell}, \frac{\delta^j}{b} \right)$ by $\frac{\delta^{i-1}}{b} + \frac{\sum_{k=i}^{j} w^k}{s_\ell} + \frac{\delta^j}{b}$ in the definition of $\mathcal{F}_i^j$. Note also that $E_a^k = +\infty$ if the algorithm fails to match the period T.

For several applications, let $E(a, k)$ the minimum energy consumed by k processors on the first applications $1, \dots, a$, so we are looking for $E(A, p)$. This energy can be computed recursively, thanks to the recurrence relation:

$$\forall k \in \{1,\dots,p\}, \forall a \in \{2,\dots,A\}, \quad E(a,k) = \min_{q \in \{0,\dots,k-1\}} \left(E_a^q + E(a-1,k-q) \right)$$

and the initialization: $\forall k \in \{1, \dots, p\},\ E(1,k) = E_1^k$.

The overall complexity is $O(AN^3p^2)$. ■

2.5.5 Period/latency/energy minimization

When mixing the three criteria, the problem becomes NP-hard even for fully homogeneous platforms, no communication, and a single application. The combinatorial nature of the problem comes from the fact that even if processors are identical, they are multi-modal and each of them may run at a different speed.

Theorem 2.15. *On fully homogeneous platforms, with a single application and without any communication cost, finding a one-to-one mapping that solves the tri-criteria problem is NP-hard.*

Proof. We consider the associated decision problem: given a period T, a latency L and an energy E, does there exist a one-to-one mapping of period less than T, latency less than L and energy less than E?

The problem is obviously in NP: given a period, a latency, an energy and a mapping, it is easy to check in polynomial time that the mapping is valid.

To establish the completeness, we use a reduction from 2-PARTITION [44]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given n strictly positive integers $a_1, a_2, \dots, a_n$, does there exists a subset I of $\{1, \dots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$. Let $K = \alpha \times S + 2$, where α is the exponent used in the computation of the energy (see Section 2.4.5).

We build an instance $\mathcal{I}_2$ of our problem with n identical processors, each with $m = 2n + 1$ modes such that:

$$\forall i \in \{1, \dots, n\} \quad \begin{cases} s_{2i-1} = K^i \\ s_{2i} = K^i + \frac{a_i X}{K^{i(\alpha-1)}} \end{cases}$$

and a pipelined application composed of n stages, with computation costs $w_i = K^{i(\alpha+1)}$.

Intuitively, the idea is to choose K such that (i) stage weights are far enough from one another; and (ii) there is a gap between (s_{2i-1}, s_{2i}) and (s_{2j-1}, s_{2j}). Then the mapping will use exactly one component of every pair (s_{2i-1}, s_{2i}).

We claim that for each $j \in \{2, \dots, n\}$, we have

$$K^{j\alpha} > \sum_{i=1}^{j-1} K^{i\alpha} + \alpha \left(\frac{S}{2} - \frac{1}{2} \right) \quad \text{and} \quad K^{j\alpha+1} > \sum_{i=1}^{j} K^{i\alpha} + \left(K^{1-\alpha} \times a_{j-1} + 1 - \frac{S}{2} \right).$$

To prove the claim, let $j \in \{2, \dots, n\}$. On the one side,

$$\begin{aligned} \sum_{i=1}^{j-1} K^{i\alpha} + \alpha \left(\frac{S}{2} - \frac{1}{2} \right) &< \sum_{i=1}^{j-1} K^{i\alpha} + \alpha S \\ &< (j-1) K^{(j-1)\alpha} + K \\ &< j K^{(j-1)\alpha} < K^{j\alpha} \,. \end{aligned}$$

On the other side,

$$\begin{aligned} \sum_{i=1}^{j} K^{i\alpha} + \left(K^{1-\alpha} \times a_{j-1} + 1 - \frac{S}{2} \right) &< \sum_{i=1}^{j} K^{i\alpha} + K^{1-\alpha} \times K \\ &< j K^{j\alpha} + K^{2-\alpha} < (j+1) K^{j\alpha} \\ &< K^{j\alpha+1} \,. \end{aligned}$$

We deduce that for each $j \in \{2, \dots, n\}$ and each $0 < X < 1$,

$$K^{j\alpha} > \sum_{i=1}^{j-1} K^{i\alpha} + \alpha X \left(\frac{S}{2} - \frac{1}{2} \right) \quad \text{and} \quad K^{j\alpha+1} > \sum_{i=1}^{j} K^{i\alpha} + X \left(K^{1-\alpha} \times a_{j-1} + 1 - \frac{S}{2} \right).$$

For all $i \in \{1, \dots, n\}$, if we choose speed s_{2i} instead of speed s_{2i-1}, the additional energy is:

$$\begin{aligned} s_{2i}^{\alpha} - s_{2i-1}^{\alpha} &= (K^i + \frac{a_i X}{K^{i(\alpha-1)}})^{\alpha} - K^{i\alpha} \\ &= K^{i\alpha}(1 + \alpha \frac{a_i X}{K^{i\alpha}} + o(X)) - K^{i\alpha} \\ &= \alpha a_i X + f_i^E(X) \ , \end{aligned}$$

where $f_i^E(X) \underset{x \to 0}{=} o(X)$.

In the same way, for each $i \in \{1, \dots, n\}$, the difference in latency when using speed s_{2i} instead of speed s_{2i-1} to execute stage $\mathcal{S}_i$ is:

$$\begin{aligned} \frac{w_i}{s_{2i-1}} - \frac{w_i}{s_{2i}} &= \frac{K^{i(\alpha+1)}}{K^i} - \frac{K^{i(\alpha+1)}}{K^i + \frac{a_i X}{K^{i(\alpha-1)}}} \\ &= \frac{K^{i(\alpha+1)}}{K^i} - \frac{K^{i(\alpha+1)}}{K^i}\left(1 - \frac{a_i X}{K^{i\alpha}} + o(X)\right) \\ &= a_i X - f_i^L(X) \ , \end{aligned}$$

where $f_i^L(X) \underset{x \to 0}{=} o(X)$.

For all $i \in \{2, \dots, n\}$, the time to execute $\mathcal{S}_i$ at speed s_{2i-2} is:

$$\begin{aligned} \frac{w_i}{s_{2i-2}} &= \frac{K^{i(\alpha+1)}}{K^{i-1} + \frac{a_{i-1} X}{K^{(i-1)(\alpha-1)}}} \\ &= \frac{K^{i(\alpha+1)}}{K^{i-1}}\left(1 - \frac{a_{i-1} X}{K^{(i-1)\alpha}} + o(X)\right) \\ &= K^{i\alpha+1} - K^{1-\alpha} \times a_{i-1} X + f^{L_i}(X) \end{aligned}$$

So we choose $X < 1$ small enough, so that for each $i \in \{1, \dots, n\}$,

$$\begin{cases} |f_i^E(X)| < X \times \frac{\alpha}{2n} \\ |f_i^L(X)| < X \times \frac{1}{2n} \end{cases}$$

and for all $i \in \{2, \dots, n\}$, $|f^{L_i}(X)| < X \times \frac{1}{2}$.

We are now ready to choose the latency, energy and period bounds. Let E^* and L^* be the energy and latency obtained when $\mathcal{S}_i$ is executed at speed s_{2i-1} for all $i \in \{1, \dots, n\}$,

$$E^* = \sum_{i=1}^{n} s_{2i-1}^{\alpha} = \sum_{i=1}^{n} K^{i\alpha} \quad \text{and} \quad L^* = \sum_{i=1}^{n} \frac{w_i}{s_{2i-1}} = E^*.$$

We ask whether it is possible to achieve an energy $E^o = E^* + \alpha X(S/2 + 1/2)$, a latency $L^o = L^* - X(S/2 - 1/2)$ and a period $T^o = L^o$.

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Assume first that $\mathcal{I}_1$ has a solution. For each $i \in I$, stage $\mathcal{S}_i$ is executed at speed s_{2i}, and for each $i \in \{1,\dots,n\} \setminus I$, stage $\mathcal{S}_i$ is executed at speed s_{2i-1}. The mapping consumes an energy E and has a latency L, where:

$$\begin{aligned}
E &= E^* + \sum_{i \in I} \left(s_{2i}^{\alpha} - s_{2i-1}^{\alpha}\right) \\
&= E^* + \sum_{i \in I} \left(\alpha a_i X + f_i^E(X)\right) \\
&\leq E^* + \sum_{i \in I} \left(\alpha a_i X + \frac{\alpha X}{2n}\right) \leq E^* + \alpha X \left(\frac{S}{2} + \frac{1}{2}\right) \\
E &\leq E^o \\
L &= L^* - \sum_{i \in I} \left(\frac{w_i}{s_{2i-1}} - \frac{w_i}{s_{2i}}\right) \\
&= L^* - \sum_{i \in I} \left(a_i X - f_i^L(X)\right) \\
&\leq L^* - \sum_{i \in I} \left(a_i X - \frac{X}{2n}\right) \leq L^* - X \left(\frac{S}{2} - \frac{1}{2}\right) \\
L &\leq L^o
\end{aligned}$$

Because $T^o = L^o$, and because we fulfill the latency constraint, we fulfill the period constraint too. We conclude that $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. We first show that for each $i \in \{1,\dots,n\}$, stage $\mathcal{S}_i$ is executed at speed either s_{2i-1} or s_{2i}. Let $(\mathcal{P}_j)$ be the property: for each $i \in \{j,\dots,n\}$, there is a single processor running at speed s_{2i-1} or s_{2i}, and this processor is assigned stage $\mathcal{S}_i$. We first prove that $(\mathcal{P}_n)$ is true. On the one hand, if two processors were running at speed s_{2n-1} or s_{2n}, they would consume an energy $E \geq 2 s_{2n-1}^{\alpha} > K^{n\alpha} + \sum_{i=1}^{n-1} K^{i\alpha} + \alpha X \left(\frac{S}{2} + \frac{1}{2}\right) > E^o$.

On the other hand, if no processor was running at speed s_{2n-1} or s_{2n}, the latency would verify

$$\begin{aligned}
L &\geq \frac{w_n}{s_{2n-2}} \geq K^{n\alpha+1} - K^{1-\alpha} \times a_{n-1} X + f^{L_i}(X) \\
&> \sum_{i=1}^{n} K^{i\alpha} + X \left(K^{1-\alpha} \times a_{n-1} + 1 - \frac{S}{2}\right) - K^{1-\alpha} \times a_{n-1} X + f^{L_i}(X) \\
&> \sum_{i=1}^{n} K^{i\alpha} - X \left(\frac{S}{2} - \frac{1}{2}\right) + \left(\frac{X}{2} + f^{L_i}(X)\right) \\
L &> L^o \, .
\end{aligned}$$

We conclude that $(\mathcal{P}_n)$ is true. We now proceed by induction. If for some $j \in \{3,\dots,n\}$, $(\mathcal{P}_j)$ is true, then we show that $(\mathcal{P}_{j-1})$ is true in a quite similar way. In the end, $(\mathcal{P}_2)$ is true (and the processor that is assigned stage $\mathcal{S}_1$ is running either at speed s_1, or at speed s_2). Let I the subset of $\{1,\dots,n\}$ such that the processor that is assigned the stage $\mathcal{S}_i$ is running at speed s_{2i}. Then for each $i \in \{1,\dots,n\} \setminus I$, the processor that is assigned stage $\mathcal{S}_i$ is running at speed s_{2i-1}. The consumed energy is $E = E^* + \sum_{i \in I} \left(\alpha a_i X + f_i^E(X)\right)$, and $E \leq E^o$, hence $\sum_{i \in I} a_i \leq \frac{S}{2} + \left(\frac{1}{2} - \frac{\sum_{i \in I} f_i^E(X)}{\alpha X}\right)$. Therefore $\sum_{i \in I} a_i < \frac{S}{2} + \left(\frac{1}{2} + \frac{1}{2}\right)$. Since the a_i are integers, we conclude that $\sum_{i \in I} a_i \leq \frac{S}{2}$.

The achieved latency is $L = L^* - \sum_{i \in I}\left(a_i X - f_i^L(X)\right)$, and $L \leq L^o$, hence

$$\sum_{i \in I} a_i \geq \frac{S}{2} - \left(\frac{1}{2} - \frac{\sum_{i \in I} f_i^L(X)}{X}\right).$$

Since $\frac{\sum_{i \in I} f_i^L(X)}{X} \leq \frac{1}{2}$, we get $\sum_{i \in I} a_i \geq \frac{S}{2}$.

Finally, $\sum_{i \in I} a_i = \frac{S}{2}$ and $\mathcal{I}_1$ has a solution, which concludes the proof. ∎

Theorem 2.16. *On fully homogeneous platforms, with a single application and without any communication cost, finding an interval mapping that solves the tri-criteria problem is NP-hard.*

Proof. We only give the sketch of the completeness proof, which reuses the proof of Theorem 2.15. To construct the instance $\mathcal{I}_2$, we insert big stages between the previous stages. We add a big speed to the processor modes, adjusted to allow the execution of exactly one big stage during the period. More formally, we build a pipeline composed of $2n - 1$ stages, such that $\forall i \in \{1, \ldots, n\}, w_{2i-1} = K^{i(\alpha+1)}$ and $\forall i \in \{1, \ldots, n-1\}, w_{2i} = K^{(n+1)(\alpha+1)}$ We use $2n - 1$ identical processors, that can run $2n + 1$ modes, such that $\forall i \in \{1, \ldots, n\}, s_{2i-1} = K^i$ and $s_{2i} = K^i + \frac{a_i X}{K^{i\alpha}}$. We also let $s_{2n+1} = K^{n+1}$.

We search for an interval mapping, whose energy does not exceed $E^o = (n-1)K^{(n+1)\alpha} + E^* + \alpha X(S/2 + 1/2)$, whose latency does not exceed $L^o = (n-1)K^{(n+1)\alpha} + L^* - X(S/2 - 1/2)$, and whose period does not exceed $T^o = K^{(n+1)\alpha}$. If the instance $\mathcal{I}_1$ of 2-PARTITION has a solution, we proceed like in the previous proof, and map every big stage onto a processor that is running in its highest mode. All constraints are fulfilled.

If the instance $\mathcal{I}_2$ has a solution, we have to run processors that are assigned a big stage in their highest mode. Moreover, these processors cannot be assigned other stages. All we have to do next is to find a one-to-one mapping of the unassigned stages, with the additional constraint that we cannot run the remaining processors in their highest modes without exceeding the energy bound. We then conclude as in the proof of Theorem 2.15. ∎

2.5.6 Summary of complexity results for the PATH model

Table 2.1 summarizes all complexity results with the PATH latency model, for one-to-one and interval mappings without resource sharing.

<table>
<tr><th rowspan="2"></th><th>proc-hom</th><th colspan="3">proc-het</th></tr>
<tr><th>com-hom</th><th>special-app</th><th>com-hom</th><th>com-het</th></tr>
<tr><td>Per - one-to-one</td><td colspan="3">polynomial (binary search)</td><td>NP-c.</td></tr>
<tr><td>Per - interval</td><td>poly (dyn. prog. + greedy)</td><td>NP-complete(*)</td><td colspan="2">NP-complete</td></tr>
<tr><td>Lat - one-to-one</td><td>polynomial</td><td colspan="2">NP-complete(*)</td><td>NP-c.</td></tr>
<tr><td>Lat - interval</td><td colspan="3">polynomial (binary search)</td><td>NP-c.</td></tr>
<tr><td>Per/Lat - both</td><td>polynomial</td><td colspan="3">NP-complete</td></tr>
<tr><td>Per/En - one-to-one</td><td colspan="3">polynomial (minimum matching)</td><td>NP-c.</td></tr>
<tr><td>Per/En - interval</td><td>poly (dyn. prog.)</td><td colspan="3">NP-complete</td></tr>
<tr><td>Per/Lat/En - both</td><td colspan="4">NP-complete</td></tr>
</table>

Table 2.1: Complexity results with the PATH latency model.

For the mono-criterion problems, most NP-completeness proofs come from the single application problem which already was NP-hard, see [17, 19] for the proofs. The two special entries denoted with (*) are problem instances which could be solved in polynomial time for a single application, but becomes NP-hard with several ones. Remaining entries correspond to polynomial algorithms that were already existing for a single application and that have been extended for several ones.

For the bi-criteria problems, we provide new polynomial algorithms to minimize one of the criterion, given a bound on the other one. NP-completeness results are obtained from the mono-criterion complexity results.

Finally, the tri-criteria problem turns out to be NP-hard even for fully homogeneous platforms, no communication and a single application.

2.6 Complexity results with the WAVEFRONT model

In the previous section, we have performed an exhaustive complexity study considering the PATH latency model, and hence restricting to mapping rules without resource sharing (one-to-one or interval mappings). We have provided new polynomial algorithms for multiple applications and results of NP-completeness. However, when considering resource sharing and general mappings, we use the WAVEFRONT latency model, as explained in the framework (see Section 2.4.4).

In this section, we investigate the impact of this model on the complexity results. Since the latency definition is now closely related to the period definition, we consider only latency in combination with period. For the period/latency combination, we minimize the latency for a fixed period. For the tri-criteria problem, both period and latency are fixed, and we minimize the energy criterion.

Also, we do not restrict the study to one-to-one and interval mappings, but also discuss general mappings. It turns out that the period minimization problem is NP-hard for such mappings, even for fully homogeneous platforms, no communication and a single application. Therefore, all multi-criteria problems with general mappings are NP-hard.

All results are summarized in Table 2.2.

2.6.1 Period minimization

All complexity results for period minimization were already established in Section 2.5.1, except for general mappings. It turns out that the problem is NP-hard for general mappings, even for fully homogeneous platforms, no communication and a single application.

Theorem 2.17. *On fully homogeneous platforms with no communication, the problem of finding a general mapping that minimizes the period of a single application is NP-complete.*

	proc-hom	**proc-het**		
	com-hom	**special-app**	**com-hom**	**com-het**
Per/* - *general*	NP-complete			
Per/Lat - *one-to-one*	polynomial			NP-complete
Per/Lat - *interval*	polynomial	NP-complete		
Per/Lat/En - *one-to-one*	polynomial			NP-complete
Per/Lat/En - *interval*	poly (dyn. prog.)	NP-complete		

Table 2.2: Complexity results with the WAVEFRONT latency model.

Proof. The reduction is straightforward, with a reduction from 2-PARTITION [44]: the application consists of n stages and there are two identical processors. Stages must be partitioned in two sets of equal computational weight, which amounts to 2-partition the stages. ■

As a corollary, all multi-criteria problems are NP-hard for general mappings, since they all involve the period criterion (because of the energy and latency definitions).

2.6.2 Period/latency minimization

With heterogeneous processors and interval mappings, we already know that the period minimization problem is NP-hard, and therefore it remains NP-hard when combining it with the latency criterion. However, the result does not hold any longer for one-to-one mappings, while the bi-criteria problem was NP-hard with the PATH latency model. Actually, with homogeneous communications, the latency of an application with n stages is always $(2n-1) \times T$, where T is the period of the application, and therefore the latency is minimized when the period is minimized. The bi-criteria problem amounts in this case to the period minimization problem, which is polynomial (binary search algorithm, see Section 2.5.1).

For homogeneous platforms, we propose below a polynomial algorithm for the period/latency/energy combination on homogeneous platforms and interval mappings. This algorithm can be used to solve the easier bi-criteria problem with no energy criterion.

2.6.3 Period/latency/energy minimization

As motivated earlier, we focus on the tri-criteria problem of minimizing energy under constraints on period ans latency. It turns out that this problem becomes polynomial for interval mappings without resource sharing on fully homogeneous platforms, while it was NP-complete with the classical definition of latency (see Theorem 2.16).

For one-to-one mappings, the problem is polynomial for **com-hom** platforms with different speed processors. Indeed, similarly to the period/latency problem, minimizing the latency is equivalent to minimizing the period for such mappings because of the WAVEFRONT latency model and the one-to-one mapping.

Theorem 2.18. *With the* WAVEFRONT *latency model, the tri-criteria problem is polynomial on fully homogeneous platforms for interval mappings without reuse.*

Proof. The optimal solution for interval mappings relies on an intricate nesting of two dynamic programming algorithms. The first one solves the problem with one single application: it recursively computes the optimal energy consumption that can be achieved by mapping one stage interval to exactly q processors. Then another dynamic programming algorithm finds the minimum energy consumption with several applications, recursively trying all possible distributions of processors to applications, and using the first algorithm to compute the optimal energy consumption for each application, given the number of processors allocated to this application.

For the single application problem, Let n be the number of stages of this application, T_{giv} be the given period, and L_{giv} be the given latency. First of all, note that the latency is given by $L = (2m-1) \times T_{\text{giv}}$, where m is the number of intervals. Therefore, we can compute a priori the maximum possible number of intervals in the mapping. Let $m^{\max}$ be this number; note that it cannot exceed n, the total number of stages, nor p, the number of processors: $m^{\max} = \min(n, p, \lfloor (\frac{L_{\text{giv}}}{T_{\text{giv}}} + 1)/2 \rfloor)$. If we use more intervals, the bound on the latency will be exceeded. Otherwise, we just have to check if the period constraint is fulfilled.

We exhibit a dynamic programming algorithm that returns the optimal energy consumption. We compute recursively the value $E(i,j,q)$, which is the optimal energy consumption that can be achieved by any interval-based mapping of stages $\mathcal{S}^i$ to $\mathcal{S}^j$ using exactly q processors. The goal is to determine $\min_{m\in\{1,\dots,m^{\max}\}} E(1,n,m)$. The recurrence relation can be expressed as:

$$E(i,j,q) = \min_{i\leq \ell \leq j-1} \left(E(i,\ell,q-1) + E(\ell+1,j,1) \right),$$

with the initializations:
– $E(i,i,q) = +\infty$ if $q > 1$ (we cannot run one stage with many processors);
– $E(i,j,1) = \begin{cases} \min \mathcal{F}^{i,j} & \text{if } \mathcal{F}^{i,j} \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$, where

$$\mathcal{F}^{i,j} = \left\{ E_{dyn}(s) + E_{stat} \mid \max\left(\frac{\delta^{i-1}}{b}, \frac{\sum_{k=i}^{j} w^k}{s}, \frac{\delta^j}{b} \right) \leq T_{\text{giv}} \right\}_{s\in \mathsf{S}}$$

Since the platform is homogeneous, we denote by E_{stat} the static energy of all processors, and by $E_{dyn}(s)$ the dynamic energy consumed at speed s ($s \in \mathsf{S}$). Then, the recurrence is easy to justify: to compute $E(i,j,q)$, we create an interval from stages $\mathcal{S}^{\ell+1}$ to $\mathcal{S}^j$ that is assigned to one single processor, and we use the $q-1$ remaining processors to process stages $\mathcal{S}^i$ to $\mathcal{S}^\ell$. The initialization states that one single stage cannot be run on exactly more than one processor, and it returns the energy consumed by the processor in charge of interval $[i,j]$ so that the bound on the period is satisfied.

With many applications, for $a \in \{1,\dots,A\}$ and $q \in \{0,\dots,p\}$, let E_a^q the minimum energy consumed by q processors on the application a, computed by one the previous dynamic programming algorithms. If the period constraint cannot be fulfilled, or if the latency constraint cannot be fulfilled ($q > k_a^{\max}$), we set $E_a^q = +\infty$.

We recursively compute the value $E(a,q)$, which is the minimum energy consumed by exactly q processors on applications $1,\dots,a$. The goal is thus to compute $\min_{1\leq q\leq p} E(A,q)$. The recurrence relation can be expressed as:

$$E(a,q) = \min_{1\leq r\leq q-1} \left(E(a-1,q-r) + E_a^r \right),$$

with the initialization:

$$E(1,q) = E_1^q, \forall 1 \leq q \leq p.$$

Indeed, when there is only one application left, the result is known from the previous dynamic programming algorithm. For several applications, we try to assign r processors to application a, and find the value of r which returns the lowest energy consumption. ■

2.7 Simulations with the WAVEFRONT model

In this section, we first propose an integer linear program which allows us to solve the tri-criteria problem under the WAVEFRONT model with or without processor reuse, even on fully heterogeneous platforms. However, this program has a prohibitive execution time for large platforms (it may run in exponential time). Therefore, we propose some polynomial-time heuristics in Section 2.7.2. For small problem instances, we evaluate the absolute performance of the heuristics with respect to the optimal solution returned by the integer linear program, while for large problem instances we have to restrict to a relative comparison of their performance (see Section 2.7.3).

2.7.1 Integer linear program

This section provides an integer linear program which gives the exact solution to the tri-criteria problem with the WAVEFRONT model. Although we expect its cost to restrict its use to small problem instances, this program allows us to assess the absolute performance of the heuristics introduced in Section 2.7.2 on these instances. The optimization problem includes parameters to describe the applications and the platform, and constraints, as for instance those on the periods. The linear program assigns variables so that they fulfill all constraints, and so that the objective function (the energy) is minimized. We observe that for a given application we can compute the maximum possible number of intervals, given the latency threshold of this application, before calling the linear program.

Parameters

Applications: For all $a \in \{1, \dots, A\}$, we note $n(a)$ the number of stages in the application a, $T(a)$ its period and $m(a)$ its maximum number of intervals. We add $2A$ fictitious stages $\mathcal{S}_1^0, \dots, \mathcal{S}_A^0$, $\mathcal{S}_1^{n(1)+1}, \dots, \mathcal{S}_A^{n(A)+1}$, respectively assigned to processors $\mathcal{P}_{\mathsf{in}_1}, \dots, \mathcal{P}_{\mathsf{in}_A}$, and $\mathcal{P}_{\mathsf{out}_1}, \dots, \mathcal{P}_{\mathsf{out}_A}$.

Stages: For all $a \in \{1, \dots, A\}$ and $k \in \{0, \dots, n(a)+1\}$, let $w(a,k)$ be the weight of stage $\mathcal{S}_a^k$, and, if $k \neq n(a)+1$, let $\delta(a,k)$ be the output data of stage $\mathcal{S}_a^k$.

Processors: We denote by $\mathcal{IO}$ the index set of input and output processors (hence $\mathcal{IO} = \{\mathsf{in}_1, \dots, \mathsf{in}_A\} \cup \{\mathsf{out}_1, \dots, \mathsf{out}_A\}$), and by $\mathcal{NIO}$ the index set of the other processors (with $|\mathcal{NIO}| = p$). We also assume that there is an order on $\mathcal{NIO} \cup \mathcal{IO}$. Each processor $\mathcal{P}_u$, for $u \in \mathcal{NIO}$, has an input (resp. output) network card capacity of $B^{in}(u)$ (resp. $B^{out}(u)$), and a static energy $E_{stat}(u)$. It can be in $m(u)+1$ different modes. Its speed in mode ℓ, where $\ell \in \{0, \dots, m(u)\}$, is $s(u,\ell)$; mode 0 corresponds to the inactivity of the processor (and thus $s(u,0) = 0$); therefore, for $\ell \in \{1, \dots, m(u)\}$, the power consumption of $\mathcal{P}_u$ in this mode is $E(u,\ell) = E_{stat}(u) + s(u,\ell)^\alpha$, while $E(u,0) = 0$ (no energy consumption when inactive). The link bandwidth between processors $\mathcal{P}_u$ and $\mathcal{P}_v$, with $(u,v) \in \mathcal{NIO}^2$ and $u \neq v$, is denoted by $b(\min(u,v), \max(u,v))$.

Variables

- For $a \in \{1, \dots, A\}$, $k \in \{0, \dots, n(a)+1\}$ and $u \in \mathcal{NIO} \cup \mathcal{IO}$, $x_{a,k,u}$ is a boolean variable equal to 1 if stage $\mathcal{S}_a^k$ is assigned to processor $\mathcal{P}_u$; we have $x_{a,0,\mathsf{in}_a} = x_{a,n(a)+1,\mathsf{out}_a} = 1$, and $x_{a,k,\mathsf{in}_a} = x_{a,k,\mathsf{out}_a} = 0$ for $a \in \{1, \dots, A\}$ and $1 \leq k \leq n(a)$. We also have $x_{a,k,\mathsf{in}_{a'}} = x_{a,k,\mathsf{out}_{a'}} = 0$ for $a \in \{1, \dots, A\}$, $0 \leq k \leq n(a)+1$ and $a' \neq a$.
- For $a \in \{1, \dots, A\}$, $k \in \{0, \dots, n(a)\}$, $(u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2$, $y_{a,k,u,v}$ is a boolean variable equal to 1 if stage $\mathcal{S}_a^k$ is assigned to $\mathcal{P}_u$ and stage $\mathcal{S}_a^{k+1}$ is assigned to $\mathcal{P}_v$. For all $u \in \mathcal{NIO} \cup \mathcal{IO}$ and $a \in \{1, \dots, A\}$, if $k \neq 0$ then $y_{a,k,\mathsf{in}_a,u} = 0$ and if $k \neq n(a)$ then $y_{k,u,\mathsf{out}_a} = 0$.
- For $u \in \mathcal{NIO}$ and $\ell \in \{0, \dots, m(u)\}$, $z_{u,\ell}$ is a boolean variable equal to 1 if processor $\mathcal{P}_u$ is in the mode ℓ and 0 otherwise.
- For $u \in \mathcal{NIO}$, $a \in \{1, \dots, A\}$ and $k \in \{1, \dots, n(a)\}$, $s_{a,k,u}$ is the computing power given by processor $\mathcal{P}_u$ to compute stage $\mathcal{S}_a^k$.
- For $(u,v) \in (\mathcal{NIO} \cup \mathcal{IO})^2$, $a \in \{1, \dots, A\}$ and $k \in \{0, \dots, n(a)\}$, $b_{a,k,u,v}$ is the allocated part of the link bandwidth between $\mathcal{P}_u$ and $\mathcal{P}_v$ so that $\mathcal{P}_u$ will send the output data of the stage $\mathcal{S}_a^k$ to $\mathcal{P}_v$.

Objective function

We aim at minimizing $E = \sum_{u=1}^{p} \sum_{\ell=0}^{m(u)} z_{u,\ell} \times E(u,\ell)$.

Constraints

- Each processor runs in one and only one mode: $\forall u \in \mathcal{NIO}, \sum_{\ell=0}^{m(u)} z_{u,\ell} = 1$.
- Each stage is assigned to a processor:
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)+1\}, \sum_{u \in \mathcal{NIO} \cup \mathcal{IO}} x_{a,k,u} = 1,$
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \sum_{(u,v)\in(\mathcal{NIO}\cup\mathcal{IO})^2} y_{a,k,u,v} = 1.$
- By construction:
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \forall (u,v) \in (\mathcal{NIO}\cup\mathcal{IO})^2, x_{a,k,u} + x_{a,k+1,v} \leq 1 + y_{a,k,u,v}.$
- Each processor does not exceed its computing speed:
  $\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=1}^{n(a)} s_{a,k,u} \leq \sum_{\ell=0}^{m(u)} z_{u,\ell} \times s(u,\ell).$
- Each processor does not exceed its maximum outgoing and ingoing total communication volume:

$$\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=1}^{n(a)} \sum_{\substack{v \in \mathcal{NIO}\cup\mathcal{IO} \\ v \neq u}} b_{a,k,u,v} \leq B^{out}(u),$$

$$\forall u \in \mathcal{NIO}, \sum_{a=1}^{A} \sum_{k=0}^{n(a)-1} \sum_{\substack{v \in \mathcal{NIO}\cup\mathcal{IO} \\ v \neq u}} b_{a,k,v,u} \leq B^{in}(u).$$

- The link capacity is not exceeded between two processors:
  $\forall u \in \mathcal{NIO}\cup\mathcal{IO}, \forall v > u, \sum_{a=1}^{A} \sum_{k=0}^{n(a)} (b_{a,k,u,v} + b_{a,k,v,u}) \leq b(u,v).$
- Computation time fits in the period (no constraint if stage $\mathcal{S}_a^k$ is not assigned to processor $\mathcal{P}_u$):
  $\forall a \in \{1,\dots,A\}, \forall k \in \{1,\dots,n(a)\}, \forall u \in \mathcal{NIO}, x_{a,k,u} \times w(a,k) \leq P(a) \times s_{a,k,u}.$
- Communication time fits in the period:
  $\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \forall u \in \mathcal{NIO}\cup\mathcal{IO}, \forall v \neq u,$
  $y_{a,k,u,v} \times \delta(a,k) \leq P(a) \times b_{a,k,u,v}.$
- The maximum number of intervals is not exceeded:
  $\forall a \in \{1,\dots,A\}, \sum_{(u,v)\in\mathcal{NIO}^2,\, u\neq v} \sum_{k=1}^{n(a)} y_{a,k,u,v} \leq m(a) - 1.$

Additional constraints for interval mappings with no reuse

The previous constraints correspond to the problem of general mappings with processor reuse. We can obtain the optimal solution for interval mappings with no reuse, adding two more constraints:

– A processor cannot process two stages of two different applications:
$$\forall a \in \{1,\dots,A\}, \forall a' \in \{1,\dots,A\} \setminus \{a\}, \forall k \in \{0,\dots,n(a)\}, \forall k' \in \{0,\dots,n(a')\},$$
$$\forall u \in \mathcal{NIO}, x_{a,k,u} + x_{a',k',u} \leq 1.$$

– A processor cannot process two different intervals of the same application:
$$\forall a \in \{1,\dots,A\}, \forall k \in \{0,\dots,n(a)\}, \forall k' \in \{k+1,\dots,n(a)\}, \forall u \in \mathcal{NIO},$$
$$\forall v \in \mathcal{NIO} \setminus \{u\}, \forall v' \in \mathcal{NIO} \setminus \{u\}, y_{a,k,u,v} + y_{a,k',v',u} \leq 1.$$

2.7.2 Heuristics

In this section, we present several heuristics for mapping streaming applications onto communication homogeneous platforms. The code of these heuristics is available at `http://graal.ens-lyon.fr/~prenaud/Codes/tri-crit.tar`.

We design three main heuristics, each of them including some variants. The first heuristic H1 is a greedy random heuristic, which will serve as a basis for comparison. The second one, H2, tries to assign each application entirely to a processor, and its variant H2-split starts either with the solution of H2 (if H2 has a solution) or assigns all applications to one processor, and then iteratively improves the current solution by splitting applications into several intervals. The last heuristic H3 changes iteratively the mode distribution until it can find a feasible mapping; the way to change the speeds comes in three variants and the way to choose the mapping comes in two variants: H3 is thus available in six variants.

Except for H2, which does not use the possibility of sharing the processors (one application onto one processor), each of the heuristic variants has two versions, with or without processor reuse, which allows us to observe the impact of resource sharing.

In several heuristics, for each processor $\mathcal{P}_u$, we keep its possible modes $(s_{u,\ell})$, for $\ell \in \{0,\dots,m_u\}$, the index ℓ_u of its current mode, and the minimum speed at which the processor must run in order to be able to process all stages that it is currently assigned to without exceeding the bound on the period, s_u^{needed}. When a stage of weight w of application a is assigned to processor $\mathcal{P}_u$, we add w/T_a to s_u^{needed}. When a stage is de-assigned, we perform a subtraction instead of the addition. Finally, the power consumption of the platform is computed from the speeds s_{u,ℓ_u}, where $s_{u,\ell_u} \geq s^{\text{needed}}(u)$ for all processors.

H1: random. At the start, each application a consists of a single interval composed of all its stages. Then we randomly draw $m_a^{\max}-1$ stages of application a, where $m_a^{\max}$ is the maximum possible number of intervals of application a such that the latency constraint is respected. Each time we draw a new stage, say $\mathcal{S}_a^k$, we create a new interval by splitting the interval containing $\mathcal{S}_a^k$ just after $\mathcal{S}_a^k$, thus generating one new interval. If a stage is drawn more than once, no new interval is created, so that the final number of intervals will lie between 1 and $m_a^{\max}$, and the latency will never be exceeded. Then we assign each interval to a random processor, without any consideration on the modes of the processors. In the "no-reuse" version of H1, a processor is assigned at most one interval, whereas in the "reuse" version, a processor can be assigned several intervals.

Finally, we decide which modes are used: for each processor we choose its first mode large enough to handle with the needed speed, if such a mode exists; if it does not, the heuristic fails. More formally, for each processor $\mathcal{P}_u$, ℓ_u is the lowest index such that $s_{u,\ell_u} \geq s_u^{\text{needed}}$ if it exists; otherwise, the heuristic fails.

Algorithm 3: H2-split(PI)

```
/* PI represents a problem instance, i.e. a
   platform/applications pair                                  */
Run H2 on PI
PIBest ← PI
repeat
    PI ← PIBest
    PIBU ← PIBest
    forall application in PI do
        if the latency authorizes a split then
            forall interval in the application do
                foreach processor p that is not assigned the interval do
                    foreach stage s in the interval do
                        Assign s to p
                        if PI is better than PIBest then
                            PIBest ← PI
                    PI ← PIBU
until PIBest is better than PI
return PIBest
```

H2: one-to-one. This heuristic assigns each application (as a single interval) to one single processor. This problem corresponds to the well-known assignment problem, and we implement the Hungarian algorithm (see [73, 43]) to solve it. The rows represent the processors, and the columns the applications. We do not have to take care of the latency constraint, because one interval by application is the best assignment from the latency perspective. For the processor $\mathcal{P}_u$ and the application a, the corresponding element of the matrix (row numbered u, column numbered a) is the smallest energy which allows the processor to run the application, if possible, and $+\infty$ otherwise.

H2-split: one-to-one with split. We first try to assign each application to one processor by calling H2. If H2 is successful, each application is assigned to one processor, and H2 finds which application to assign to which processor. We perform this assignment. The processors that are not assigned to any application are set in their mode 0. If H2 fails, we assign all application stages to the first processor of the list. If it has enough speed to execute all applications within the period bound, then ℓ_u is the smallest mode such that $s_{u,\ell_u} \geq s_u^{\text{needed}}$. Otherwise, we set $\ell_u = m_u$, but the period is not satisfied in this case.

Therefore, at this point, all the stages are assigned (and we can consider, if the applications are concatenated, that each processor is assigned an "interval"), but this mapping may not be valid: there might be a processor $\mathcal{P}_u$ such that $s_{u,\ell_u} < s_u^{\text{needed}}$.

The main idea of this heuristic is then to try to split each "interval" at any place, and to keep the best split. More precisely, a split consists in:

1. de-assigning one part of the concerned "interval";
2. assigning it to another processor $\mathcal{P}_{u'}$;
3. updating the two concerned modes ℓ_u and $\ell_{u'}$ as mentioned previously, thanks to the new values s_u^{needed} and $s_{u'}^{\text{needed}}$.

Then we have to define a way to sort the different resulting mappings in order to choose the best

one. The first thing we expect from a mapping is that it respects the period and latency bounds; once we have valid mappings with respect to these performance criteria, the best one is the one whose power consumption is the lowest. Finally, when two mappings lead to the same power consumption, we choose the one in which we are likely to spare the most energy giving the less speed to the new processor during the next split. This is why we finally sort the mappings by:

1. increasing $\sum_{u=1}^{p} \max(s_u^{\text{needed}} - s_{u,\ell_u}, 0)$ (the mapping is valid if and only if this value is equal to 0);
2. increasing energy of the platform, that is increasing $E = \sum_{u \in \{1,\ldots,p\}} E(u, \ell_u)$;
3. decreasing

$$\max \left\{ \frac{E(u,\ell_u) - E(u,\ell_u - 1)}{s_u^{\text{needed}} - s_{u,\ell_u - 1}} | u \in \{1,\ldots,p\}, \ell_u \neq 0 \right\}.$$

While we find a better mapping, we try another split. More formally, the heuristic is detailed in Algorithm 3. In the "no-reuse" version, the processor added in a split cannot be assigned another non-adjacent interval, whereas there is no constraint in the "reuse" version.

H3: increasing speeds. We start with all processors in their smallest mode. Then we map applications onto the current platform (Algorithm 4), and check whether the mapping is valid or not. If the algorithm returns true, then we are done. Otherwise, we repeatedly change the distribution of the modes and call Algorithm 4 until we find a valid mapping. There are different ways to change the distribution of the modes, thus leading to different variants of the heuristic (see below for variants *s*peed, *e*nergy and *u*pDown).

We briefly explain Algorithm 4: the mapping procedure is quite different from that of previous heuristics. Indeed, we never assign a stage to a processor if it has not enough speed to run it while not exceeding the bound on the period. In other words, H3 never allows $s_{u,\ell_u} < s_u^{\text{needed}}$. In the previous heuristics, we first decided for the mapping, and then we chose the modes. In H3, we first choose the mode of each processor, and then we try to find an assignment which is valid with these modes, and may either success or fail.

Algorithm 4: H3-mapping

```
for a ← 1 to A do
    h_a ← Σ_{i=1}^{n_a} w_a^i / k_a^max
Sort the applications by decreasing h_a in L
forall application a in L do
    k ← k_a^max
    Sort the processors by decreasing remaining speed
    while all stages are not assigned and k > 0 and the processors list is not empty do
        Assign the longest interval from the first unassigned stage to the first processor
        Remove the first processor from the list
        k ← k − 1
    if all stages are assigned then
        De-assign the last interval and assign it to the last possible processor
    else
        return false
return true
```

Algorithm 5: H3-sort-mapping

```
for a ← 1 to A do
    h_a ← Σ_{i=1, S_a^i is unassigned}^{n_a} w_a^i / k_a^max
Sort the applications by decreasing h_a in L
while L is not empty do
    Pick and remove the first application a in L
    k_a^max ← k_a^max − 1
    Sort the processors by decreasing remaining speed
    Assign the longest interval from the first unassigned stage to the first processor
    if all stages are assigned then
        De-assign the last interval and assign it to the last possible processor
    else
        if k_a^max = 0 then
            return false
        else
            Update h_a and place the application a in L
return true
```

H3-sort: application sorting. This heuristic proposes a modification in the H3-mapping procedure, in which we re-sort the applications after each interval assignment. In H3, we first sort all the applications, and, application by application, we choose a processor and assign it the longest possible interval. If all stages are not assigned, we choose another processor and try to assign the next stages, until the whole application is assigned. In H3-sort (see Algorithm 5), after the first interval assignment, we find the new place of the application in the sorted list, considering this application as if the assigned stages would not exist and if there would be one less possible interval in the application (for the latency constraint). Then we iterate.

This heuristic also comes with variants in the way of changing the distribution of modes.

H3-speed/energy/upDown. We detail now the three variants, used for both H3 and H3-sort:

- **speed:** the processors are sorted by increasing speed of the current mode (and if there is a tie by increasing speed gain between the current mode and the next higher one). We check whether we find a mapping; if yes, we stop, and if not, we upgrade the first processor (in the previous order) and repeat.
- **energy:** the processors are sorted by increasing energy spent (which is different from an ordering based on modes because of static energy). Again, if there is a tie, we refine the sort according to increasing speed gain between the current mode and the next higher one. We stop when, after upgrading, function H3 returns true.
- **upDown:** We use the same ordering of processors as in the "energy" variant, but we improve the upgrade. The main idea is that if processor modes are distant from each other, the total available speed increases a lot at each upgrade. In this variant we ensure that the total available speed is increasing at each step, but try to increase it slowly. To do that, before every upgrade, we downgrade the mode of the last upgraded processor, if the total available speed is still increasing.

Summary of heuristics. Each heuristic is denoted by its heuristic number, followed by variants. For instance, H3-sort-speed is the H3-sort heuristic with the speed variant. Also, we add "-n" at the end of the heuristic name for the "without reuse" version of the heuristic, and "-r" for the "with reuse" version. Thus, H2-split-n is the H2-split heuristic with no reuse.

Finally we consider another heuristic, called the "best" heuristic, which simply takes the minimum energy returned by all the heuristics. Of course this value is achieved by different heuristics over all simulations, but it helps quantify what can be achieved in polynomial time vs. the linear program.

2.7.3 Simulation results

We have performed a comprehensive set of simulations in order to: (i) assess the absolute performance of the heuristics, (ii) analyze the impact of reusing resources (interval vs. general mappings), and (iii) study the scalability of the heuristics. We run two simulations for each of these goals.

In the first two simulations, we compare the heuristics with the linear program that finds the optimal general mapping (denoted as cplex-r), whereas in the following two ones, we use the linear program in its "without reuse" version (cplex-n). In both cases, since the integer linear program runs in exponential time and can be very time consuming, we restrict the simulations to a small set of small platforms. On the contrary, we do not launch the linear program for the last two simulations, which allows us to deal with larger applications and platforms.

Simulation setup

We first present the simulation setup for the first four simulations, in which we run the linear program, and finally we describe the last two ones, in which we run only the heuristics.

With the linear program

In each simulation, we generate a set of 30 random platforms and applications. In Simulation 1, we explore the behavior of the heuristics when the number of possible intervals is increasing, while in Simulation 2, we increase the number of processors, in order to confirm that the (best) heuristics stay close to the optimal solution. In Simulations 3 and 4, we respectively vary the maximum static energy and the average gap between two consecutive modes in order to observe the impact of resource sharing.

For each platform, and each value of the parameter that we vary, we run all heuristics, and compute the solution of the linear program using the CPLEX software [38]. Then, for each value of the parameter, and for each heuristic, we sum up (over the platforms) the inverse of the consumed energy returned by the heuristic. If the heuristic fails, we add zero. We plot on a graph the latter sum as a function to the changing parameter. So the higher the curve, the better the heuristic. Finally, we normalize each plot by the optimal solution returned by the linear program. In other words, we show the gap that separates each heuristic from the optimal solution.

Platform sizes are chosen so that the optimal solution can be found in reasonable time (each graph has been obtained within a week, and the execution time of each heuristic was under 1 second per trial). Unless mentioned otherwise, we use those following settings in the simulations. We have 3 applications, each composed of 5 to 11 stages, whose weights vary from 5 to 9. The communication costs between stages are also ranging from 5 to 9. The latency threshold is such that 3 intervals are allowed within each application. The platform consists in 6 to 8 processors, and each of these processors has between 2 and 8 different modes. The distribution of the modes is a Gaussian law centered in 5, and the speeds are chosen between 0 and 50. The static energy of each processor is randomly drawn between 0 and 200.

In Simulation 2, we have only 2 applications with 9 to 15 stages each, and the speeds are drawn between 0 and 90, so that one processor can compute all stages. In Simulation 3, only 4 to 6 processors are available, because the problem becomes untractable starting from 7 processors. Processor speeds are drawn between 0 and 80. In these simulations, we do not represent the "sort" variant of H3, because it leads to negligible variations compared to H3.

Without the linear program

In each of these large-scale simulations, we generate a set of 5000 random platforms, since the running time of the heuristics is negligible. Simulation 5 illustrates the global behavior of the heuristics when the number of applications and processors increases, whereas Simulation 6 studies more precisely their characteristics for some large instances.

In Simulation 5, each application is composed of 15 stages, whose characteristics are the same as previously, 3 intervals are authorized within an application, and the processors have 8 modes distributed between 0 and 80. The applications of Simulation 6 are defined similarly, but this time, the processors have 10 modes, distributed between 0 and 100. For each trial, we draw between 8 and 13 applications, and between 30 and 40 processors.

Comparison with the optimal solution

Simulation 1: Latency

In this first simulation, we vary the latency of the applications: at the beginning, the latency constraint imposes that each application is mapped as a single interval, while it can go up to four in the end. All the heuristics are run in their "with reuse" variants. This simulation gives us a first idea of the ordering of the heuristics: the "upDown" is the best variant of the heuristic H3, before "energy" and "speed". The "speed" variant is the only one which is not better with fewer intervals by application, because it does not choose the processors whose static energy is low.

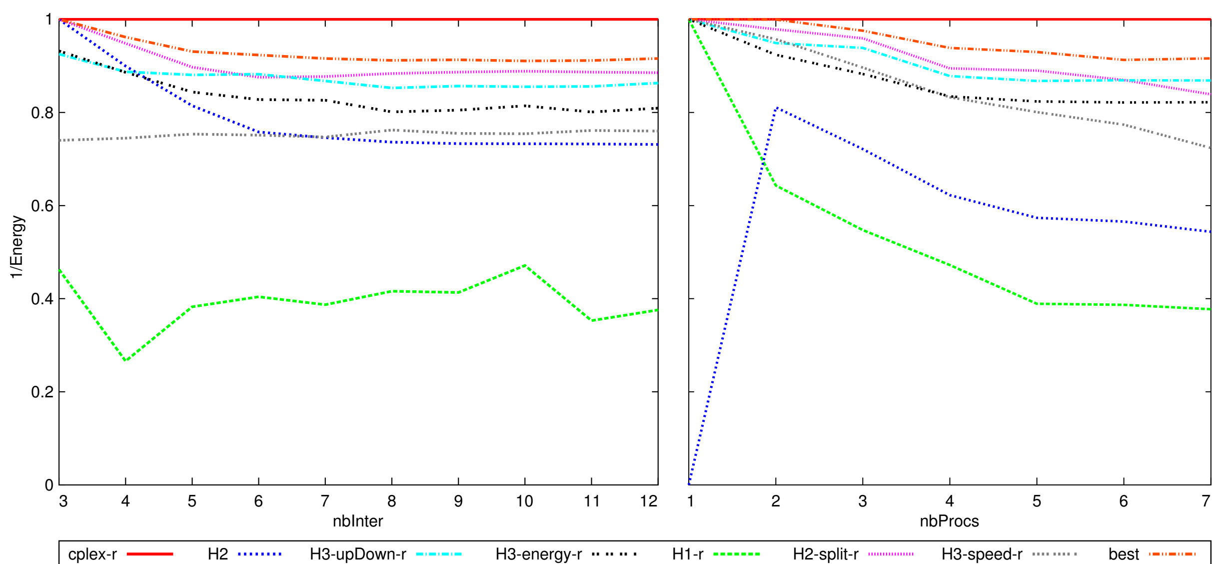


Figure 2.2: Simulation 1 and 2

Heuristic H2-split is the best heuristic on average, but for some platforms, H3-upDown is better. The best heuristic is always at 0.9 of the optimal solution. As expected, H2 finds the optimal solution when one single interval is authorized in each application. Then its performance decreases as soon as two intervals are allowed in each application. Finally, it remains approximately constant at 0.7 of the optimum. Without much surprise, heuristic H1 is worse than the others, therefore demonstrating that a random approach does not provide satisfying results.

Simulation 2: Processor number

In this second simulation, we increase the number of processors for a given application. H2 does not reuse processors, thus it does not find the solution with one processor. Then, with more than two processors, its efficiency decreases when the number of processors increases. As in the first simulation, H3-upDown-r and H2-split-r return the best results, depending upon the platform. Moreover H2-split-r is the best in average if and only if the processor number is not greater than 6. However, the “energy” and “speed” variants of H3 are always worse than H2-split-r in average. The “speed” variant becomes very bad, because when the number of available processors is increasing, these processors are used in their lowest mode, and the static part of the energy becomes crucial. Finally, the “best” heuristic is quite good, never below 0.92 of the optimal.

Impact of reuse

In this second set of simulations, we compare the heuristics to the optimal solution without reuse, in order to assess the impact of reuse on the mapping.

Simulation 3: Static energy

In this third simulation, we vary the maximum static energy, that can be drawn from 0 to 2400. When the static energy is becoming high, it is advantageous to use fewer processors. For variants “without reuse”, this leads to one processor per application. That is why H2 and H2-split-n seem to tend to the optimal solution without reuse, when the maximum static energy is increasing.

Processor reuse becomes interesting as soon as the maximum static energy exceeds 400, since the heuristics with reuse perform better than the optimal solution with no reuse. H2-split-r and H3-upDown-r are becoming more and more efficient when the static energy increases, and H2-split-r ultimately reaches 1.15 of the optimal solution without reuse for a static energy of 2400. Processor reuse allows the heuristics to use fewer processors than applications, and thus to spare some static energy cost.

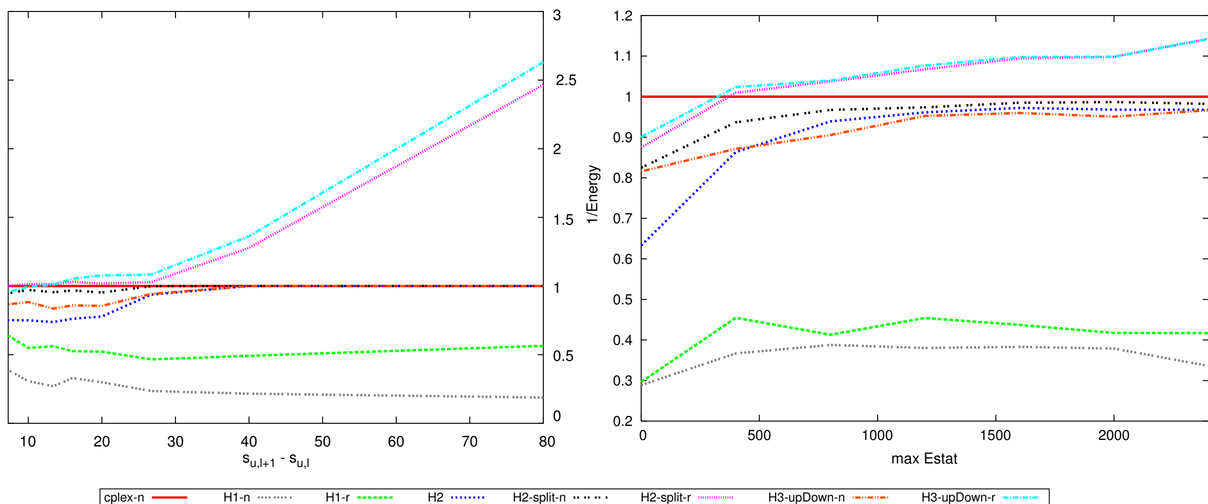


Figure 2.3: Simulation 3

Simulation 4: Mode distribution

In this fourth simulation, we vary the average gap between two modes from 5 to 40. When the modes are not close together, the first mode is high, and the best solution for the "without reuse" variants is reached with one processor per application. As before, H2 and H2-split-n tend to the optimal solution without reuse. This time, as the processors are not very different, H3-upDown-n also gets very close to the optimal solution without reuse.

The heuristics with reuse obtain much better results, in particular when the difference between modes is large. H3-upDown-r is constantly increasing and it is 2.6 times better than the optimal solution without reuse at the end. When the modes are very close, H3-upDown-r is not as competitive as the optimal solution without reuse, but it is still at 0.95. H2-split-r is almost as efficient as H3-upDown-r, and remains better than the optimal solution without reuse when the modes become closer.

More generally, resource sharing becomes interesting when the modes are not close to each other: the reuse allows us to fill up the high modes with stages of different applications.

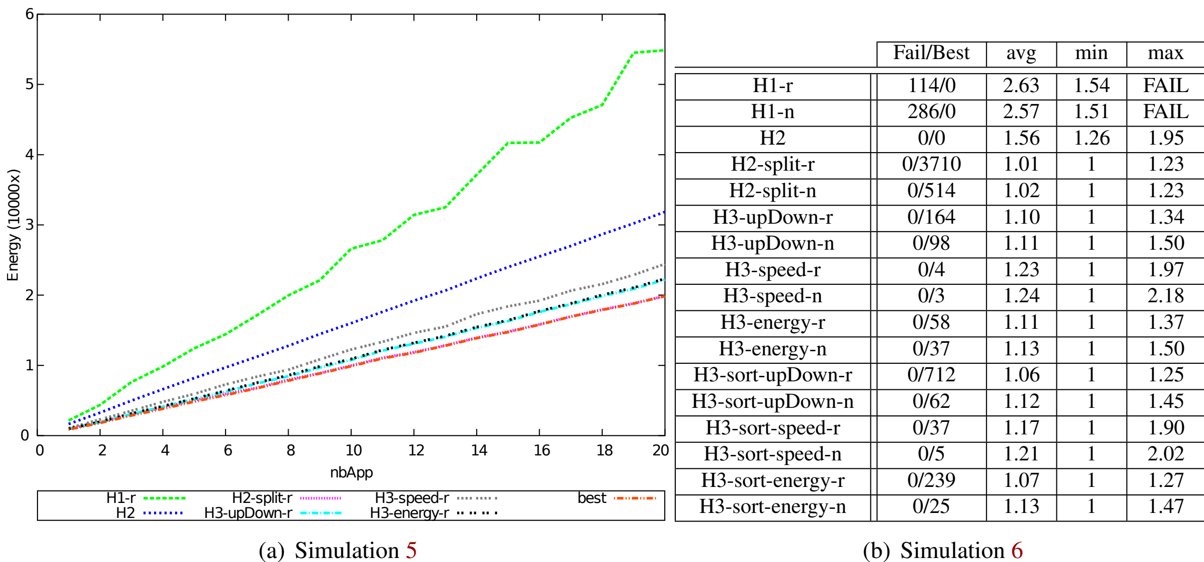
Scalability

In this last set of simulations, we study the heuristics when the instances are bigger. For such real-life instances of the problem, the integer linear program cannot be used any more, due to its high complexity.

Simulation 5: Global increase

In the fifth simulation, we increase the number of processors with the number of applications, such that there are four times more processors than applications. This time, we represent the energy on the y-axis instead of its inverse, since we cannot normalize the plots with the optimal solution anymore. Therefore, the lower the plot the better the heuristic.

H2-split-r is the best on all platforms when there are many applications, before H3-upDown-r and H3-energy-r, which almost have the same efficiency, and H3-speed-r. The more applications, the better H2-split-r, compared to the other heuristics. But for 20 applications, all heuristics execute in less than 1 second, against 3 minutes for H2-split-r.



	Fail/Best	avg	min	max
H1-r	114/0	2.63	1.54	FAIL
H1-n	286/0	2.57	1.51	FAIL
H2	0/0	1.56	1.26	1.95
H2-split-r	0/3710	1.01	1	1.23
H2-split-n	0/514	1.02	1	1.23
H3-upDown-r	0/164	1.10	1	1.34
H3-upDown-n	0/98	1.11	1	1.50
H3-speed-r	0/4	1.23	1	1.97
H3-speed-n	0/3	1.24	1	2.18
H3-energy-r	0/58	1.11	1	1.37
H3-energy-n	0/37	1.13	1	1.50
H3-sort-upDown-r	0/712	1.06	1	1.25
H3-sort-upDown-n	0/62	1.12	1	1.45
H3-sort-speed-r	0/37	1.17	1	1.90
H3-sort-speed-n	0/5	1.21	1	2.02
H3-sort-energy-r	0/239	1.07	1	1.27
H3-sort-energy-n	0/25	1.13	1	1.47

(a) Simulation 5 (b) Simulation 6

Figure 2.4: Scalability

Simulation 6: Complete comparison

In this last simulation, we study all heuristics for some large problem instances. The main characteristics of the heuristics are shown in Figure 2.4(b). We report the number of failures in the first column, and how many times the concerned heuristic has been the best one in the second column. For the last four columns, we normalize the power consumption found by each heuristic by the power consumption found by the best one and analyze the table of normalized power. The average is computed with the platforms for which the heuristic found a solution. The column "max" represents the worst case for each heuristic, this is why there is no numeric value for the heuristics which failed.

The random heuristics are the only ones which fail on some drawn platforms, and, as expected, they have the largest variability. H2-split-r is clearly the best heuristic: it finds about four times out of five a better solution than the other heuristics, and when it does not, it is not so bad, because it is on average at 0.8% of the best solution.

The variants "sort" of H3 are significantly better than the regular ones. H3-sort-upDown-r finds the best solution more often than H2-split-n, but it is worse on average. Because they do not evaluate the static energy of the processors, the variants of H3-speed do not avoid the processors with high static energy, thus they have a bigger variability and a worse average than the other variants of H3.

2.8 Conclusion

In this chapter, we have studied the problem of mapping concurrent applications onto computational platforms according to three criteria: period, latency and energy. We restricted the study to the class of applications that have a pipeline structure, and we established the complexity of the problems for different variants of mapping strategies (one-to-one, interval and general mappings), and different types of platforms (ranking from fully homogeneous to fully heterogeneous).

First we focused on one-to-one and interval mappings with no resource sharing. We considered performance criteria, namely period or latency minimization. From this study of mono-criterion problems, one striking result is the impact of having multiple concurrent applications on the problem complexity. Indeed, when several applications are in competition for resources, the period minimization problem turns out NP-hard for interval mappings with heterogeneous processors, homogeneous pipelines and without communication, while a polynomial algorithm had been found to solve the same problem with a single application. The same phenomenon happens for latency minimization with one-to-one mappings. For other period or latency minimization problems, either we were able to extend polynomial algorithms for the single application case, or the problem remained NP-complete. Considering bi-criteria problems, we were able to derive nice sophisticated multi-criteria polynomial algorithms, through the construction of bipartite graphs or the use of dynamic programming. Trade-offs were found to allow for an efficient albeit energy-aware execution. Finally, the most challenging tri-criteria problem period/latency/energy turned out to be NP-hard even with a single application on a fully homogeneous platform and no communication cost.

In order to handle processor sharing, we explained why it was mandatory to use a simpler model for the latency, and we discussed the use of the WAVEFRONT latency model. Thanks to a combination of two dynamic programming algorithms, we showed that finding an optimal interval mapping without reuse on fully homogeneous platforms can be done in polynomial time, while the same problem was shown to be NP-complete with the classical definition of latency. However, finding an optimal general mapping on any platform type, or finding any optimal interval mapping on speed-heterogeneous platforms, are NP-complete problems.

We believe that this exhaustive complexity analysis provides a solid theoretical foundation for the

study of multi-criteria mappings of several concurrent applications, in particular when combining performance and energy optimization criteria.

On the practical side, we designed several heuristics, as well as an integer linear program to compute the optimal solution (either interval-based or general) in possibly exponential time, for the WAVEFRONT latency model. The comparison of heuristics with and without processor sharing does confirm that sharing is most useful when: (i) the modes are not close to each other; and (ii) the static energy is high.

As future work, we envision to add replication to the mapping rules: a stage could be mapped onto several processors, each in charge of different data sets, in order to improve the period. This problem, partially investigated in [18], would become even more challenging in a framework accounting for energy issues. Also, it would be interesting to include the consumption induced by memory, disks, fans, and other devices, in the energy model. Finally, we would like to consider different application settings, as for instance applications that share some data paths. In this case, we expect the impact of resource sharing to be even more important, since mapping two such applications on the same resource may further reduce their period and latency.

Chapter 3

Replica placement and update strategies in tree networks

3.1 Introduction

In the two previous chapters, we have discussed how to schedule tasks onto a platform, so that the processors minimize their consumed power when they run the tasks, while guarantiing some performance criteria for the application. In this chapter, we rather consider that clients are issuing a set of requests, that are to be handled by servers. Therefore, we revisit the well-known replica placement problem in tree networks [36, 121, 16], with two new objectives: reusing pre-existing replicas, and enforcing an efficient power management. In a nutshell, the replica placement problem is the following: we are given a tree-shaped network where clients are periodically issuing requests to be satisfied by servers. The clients are known (both their position in the tree and their number of requests), while the number and location of the servers are to be determined. A client is a leaf node of the tree, and its requests can be served by one internal node. Note that the distribution tree (clients and nodes) is fixed in the approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery (see [66, 36, 82] and additional references in [121]). The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide a hierarchical and distributed access to replicas of the original data.

In the original problem, it is not a matter of energy and there is no replica before execution; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit. Nodes equipped with a replica, also called servers, serve all the clients located in their subtree (so that the root, if equipped with a replica, can serve any client). The rule of the game is to assign replicas to nodes so that the total number of replicas is minimized. This problem is well-understood: it can be solved in time $O(N^2)$ (dynamic programming algorithm of [36]), or even in time $O(N \log N)$ (optimized greedy algorithm of [121]), where N is the number of nodes.

The first contribution of this chapter is to extend replica placement algorithms to cope with power consumption constraints. To help reduce power dissipation, multi-modal processors are used: each processor has a discrete number of predefined speeds (or modes), which correspond to different voltages that the processor can be subjected to. An important result of this chapter is that minimizing power consumption is a NP-complete problem, even without pre-existing replicas, and without static power: balancing server modes across the tree already is a hard combinatorial problem.

Another contribution of this chapter is to tackle the replica placement problem when the tree is equipped with pre-existing replicas before execution. This extension is a first step towards dealing with dynamic replica management: if the number and location of client requests evolve over time,

the number and location of replicas must evolve accordingly, and one must decide how to perform a configuration change (at what cost?) and when (how frequently reconfigurations should occur?) A first approach to this complicated dynamic problem is provided in [107], where replicas are either moved or created at "regular intervals", whose duration is determined by the arrival rate of client requests. The algorithms in [107] provide a heuristic solution to the problem, but no complexity result is presented. Similarly, [33, 98, 99] tackle the problem of placing replicas with server capacity constraint, where servers are re-allocated to new sites when a performance metric degrades significantly. However, in these papers, the distribution tree is not fixed, which renders all problems highly combinatorial, and which departs from our fixed network assumption. In the present work, the aim is to assess the difficulty of a single reconfiguration, and we provide an optimal polynomial algorithm to minimize the cost of such a reconfiguration. The main difficulty here is to trade-off between two conflicting goals, namely (i) reusing existing servers rather than creating new ones, and (ii) load-balancing the requests equally among the servers.

The cost of the best power-efficient solution may be prohibitive, which calls for a bi-criteria approach: minimizing power consumption while enforcing a threshold cost that cannot be exceeded. We investigate the case where there is only a fixed number of modes and show that there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. This result has a great practical significance, because state-of-the-art processors can only be operated with a restricted number of voltage levels, hence with a few modes [60, 57].

Finally, we run simulations to show the practical utility of our algorithms, despite their high worst-case complexity. We illustrate the impact of taking pre-existing servers into account, and how power can be saved thanks to the optimal bi-criteria algorithm.

The rest of the chapter is organized as follows. Section 3.2 is devoted to a detailed presentation of the target optimization problems, and provides a summary of new complexity results. The next two sections are devoted to the proofs of these results: Section 3.3 deals with computing the optimal cost of a solution, with pre-existing replicas in the tree, while Section 3.4 addresses all power-oriented problems. Then we report the simulation results in Section 3.5. Finally, we state some concluding remarks and future working directions in Section 3.6.

3.2 Framework

This section is devoted to a precise statement of the problem. We start with the general problem without power consumption constraints, and next we introduce the power consumption model. Then we state the objective functions (with or without power), and the associated optimization problems. Finally we give a summary of all complexity results that we provide in the chapter.

3.2.1 Replica servers

We consider a distribution tree whose nodes are partitioned into a set of clients $\mathcal{C}$, and a set of N nodes, $\mathcal{N}$. The clients are leaf nodes of the tree, while $\mathcal{N}$ is the set of internal nodes. Each client $i \in \mathcal{C}$ (leaf of the tree) is sending r_i requests per time unit to a database object. Internal nodes equipped with a replica (also called *servers*) will process all requests from clients in their subtree. An internal node $j \in \mathcal{N}$ may have already been provided with a replica, and we let $\mathcal{E} \subseteq \mathcal{N}$ be the set of pre-existing servers. Servers in $\mathcal{E}$ will be either reused or deleted in the solution. Note that it would be easy to allow *client-server* nodes which play both the rule of a client and of an internal node (possibly a server), by dividing such a node into two distinct nodes in the tree.

Without power consumption constraints, the problem is to find a *solution*, i.e., a set of servers capable of handling all requests, that minimizes some cost function. We formally define a valid solution before detailing its cost. We start with some notations. Let r be the root of the tree. If $j \in \mathcal{N}$, then $\mathsf{children}_j \subseteq \mathcal{N} \cup \mathcal{C}$ is the set of children of node j, and $\mathsf{subtree}_j \subseteq \mathcal{N} \cup \mathcal{C}$ is the subtree rooted in j, excluding j. A solution is a set $\mathcal{R} \subseteq \mathcal{N}$ of servers. Each client i is assigned a single server $\mathsf{server}_i \in \mathcal{R}$ that is responsible for processing all its r_i requests, and this server is restricted to be the first ancestor of i (i.e., the first node in the unique path that leads from i up to the root r) equipped with a server (hence the name *closest* for the request service policy). Such a server must exist in $\mathcal{R}$ for each client. In addition, all servers are identical and have a limited capacity, i.e., they can process a maximum number W of requests. Let req_j be the number of requests processed by $j \in \mathcal{R}$. The capacity constraint writes

$$\forall j \in \mathcal{R}, \ \ \mathsf{req}_j = \sum_{i \in \mathcal{C} \mid j = \mathsf{server}_i} r_i \leq W. \tag{3.1}$$

Now for the cost function, because all servers are identical, the cost of operating a server can be normalized to 1. When introducing a new server, there is an additional cost create, so that running a new server costs $1 + \mathsf{create}$ while reusing a server in $\mathcal{E}$ only costs 1. There is also a deletion cost delete associated to deleting each server in $\mathcal{E}$ that is not reused in the solution. Let $E = |\mathcal{E}|$ be the number of pre-existing servers. Let $R = |\mathcal{R}|$ be the total number of servers in the solution, and $e = |\mathcal{R} \cap \mathcal{E}|$ be the number of reused servers. Altogether, the cost is

$$\mathsf{cost}(\mathcal{R}) = R + (R - e) \times \mathsf{create} + (E - e) \times \mathsf{delete}. \tag{3.2}$$

This cost function is quite general. Because of the create and delete costs, priority is always given to reusing pre-existing servers. If $\mathsf{create} + 2 \times \mathsf{delete} < 1$, priority is given to minimizing the total number of servers R: indeed, if this condition holds, it is always advantageous to replace two pre-existing servers by a new one (if capacities permit).

3.2.2 With power consumption modes

With power consumption constraints, we assume that servers may operate under a set of different speeds, or *modes*, $\mathcal{M} = \{W_1, \ldots, W_M\}$, depending upon the number of requests that they have to process per time unit. Here modes are indexed according to increasing values, and $W_M = W$, the maximal capacity. If a server $j \in \mathcal{R}$ processes req_j requests, with $W_{i-1} < \mathsf{req}_j \leq W_i$, then it is operated at mode W_i, and we let $\mathsf{mode}(j) = i$. The power consumption of a server $j \in \mathcal{R}$ obeys the classical model

$$\mathcal{P}(j) = \mathcal{P}^{(\mathsf{static})} + W^{\alpha}_{\mathsf{mode}(j)}.$$

Here, $\mathcal{P}^{(\mathsf{static})}$ is the static power consumption (constant part), while $W^{\alpha}_{\mathsf{mode}(j)}$ is the dynamic part that depends upon the operated mode. Finally, we restrict α to be a rational constant in $[2..3]$ that depends upon the model for power [63, 95, 24, 12, 30]. Note that the classical value $\alpha = 3$, used in Chapter 1 and in the simulations of Chapter 2 belongs to this interval. The total power consumption $\mathcal{P}(\mathcal{R})$ of the solution is the sum of the power consumption of all server nodes:

$$\mathcal{P}(\mathcal{R}) = \sum_{j \in \mathcal{R}} \mathcal{P}(j) = R \times \mathcal{P}^{(\mathsf{static})} + \sum_{j \in \mathcal{R}} W^{\alpha}_{\mathsf{mode}(j)}. \tag{3.3}$$

Intuitively, this equation calls for balancing two conflicting terms: static power is minimized with few servers, while dynamic power is minimized with many servers operated in the slowest mode.

With different power modes, it is natural to refine the cost function, and to include a cost for changing the mode of a pre-existing server (upgrading it to a higher mode, or downgrading it to a lower mode). In the most detailed model, we would introduce:

– create_i, the cost for creating a new server operated at mode W_i;
– $\mathsf{changed}_{i,i'}$, the cost for changing the mode of a pre-existing server from W_i to $W_{i'}$; and
– delete_i, the cost for deleting a pre-existing server operated at mode W_i.

Note that it is reasonable to let $\mathsf{changed}_{i,i} = 0$ (no change); values of $\mathsf{changed}_{i,i'}$ with $i < i'$ correspond to upgrade costs, while values with $i' < i$ correspond to downgrade costs. In accordance with these new cost parameters, given a solution $\mathcal{R}$, we count the number of servers as follows:

– n_i, the number of new servers operated at mode W_i;
– $e_{i,i'}$, the number of reused pre-existing servers whose operation modes have changed from W_i to $W_{i'}$; and
– k_i, the number of pre-existing server operated at mode W_i that have not been reused.

The cost of the solution $\mathcal{R}$ with a total of $R = \sum_{i=1}^{M} n_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} e_{i,i'}$ servers becomes:

$$\mathsf{cost}(\mathcal{R}) = R + \sum_{i=1}^{M} \mathsf{create}_i \times n_i + \sum_{i=1}^{M} \mathsf{delete}_i \times k_i + \sum_{i=1}^{M} \sum_{i'=1}^{M} \mathsf{changed}_{i,i'} \times e_{i,i'}. \tag{3.4}$$

Of course, this complicated cost function can be simplified to make the model more tractable; for instance all creation costs create_i can be set identical, all deletion costs delete_i can be set identical, all upgrade and downgrade values $\mathsf{changed}_{i,i'}$ can be set identical, and the latter can even be neglected.

3.2.3 Objective functions

Without power consumption constraints, the objective is to minimize the cost, as defined by Equation (3.2). We distinguish two optimization problems, either with pre-existing replicas in the tree or without:

– MINCOST-NOPRE, the classical cost optimization problem [36] without pre-existing replicas. Indeed, in that case, Equation (3.2) reduces to finding a solution with the minimal number of servers.
– MINCOST-WITHPRE, the cost optimization problem with pre-existing replicas.

With power consumption constraints, the first optimization problem is MINPOWER, which stands for minimizing power consumption. But the cost of the best power-efficient solution may be prohibitive, which calls for a bi-criteria approach: MINPOWER-BOUNDEDCOST is the problem to minimize power consumption while enforcing a threshold cost that cannot be exceeded. This bi-criteria problem can be declined in two versions, without pre-existing replicas (MINPOWER-BOUNDEDCOST-NOPRE) and with pre-existing replicas (MINPOWER-BOUNDEDCOST-WITHPRE).

3.2.4 Summary of results

In this chapter, we prove the following complexity results for a tree with N nodes:

Theorem 3.1. MINCOST-WITHPRE can be solved in polynomial time with a dynamic programming algorithm whose worst case complexity is $O(N^5)$.

Theorem 3.2. MINPOWER is NP-complete.

Theorem 3.3. With a constant number M of modes, both versions of MINPOWER-BOUNDEDCOST can be solved in polynomial time with a dynamic programming algorithm. The complexity of this algorithm is $O(N^{2M+1})$ for MINPOWER-BOUNDEDCOST-NOPRE and $O(N^{2M^2+2M+1})$ for MINPOWER-BOUNDEDCOST-WITHPRE.

Note that MINPOWER remains NP-complete without pre-existing replicas, and without static power: the proof of Theorem 3.2 (see Section 3.4.2) shows that balancing server modes across the tree already is a hard combinatorial problem. On the contrary, with a fixed number of modes, there are polynomial-time algorithms capable of optimizing power for a bounded cost, even with pre-existing replicas, with static power and with a complex cost function. These algorithms can be viewed as pseudo-polynomial solutions to the MINPOWER-BOUNDEDCOST problems.

3.3 Complexity results: update strategies

In this section, we focus on the MINCOST-WITHPRE problem: we need to update the set of replicas in a tree, given a set of pre-existing servers, so as to minimize the cost function.

In Section 3.3.1, we show on an illustrative example that the strategies need to trade-off between reusing resources and load-balancing requests on new servers: the greedy algorithm proposed in [121] for the MINCOST-NOPRE problem is no longer optimal. We provide in Section 3.3.2 a dynamic programming algorithm which returns the optimal solution in polynomial time, and we prove its correctness. The analysis of the execution time is given in Section 3.3.3.

3.3.1 Running example

We consider the example of Figure 3.1. There is one pre-existing replica in the tree at node B, and we need to decide whether to reuse it or not. For taking decisions locally at node A, the trade-off is the following:

– either we keep server B, and there are 7 requests going up in the tree from node A;
– either we remove server B and place a new server at node C, hence having only 4 requests going up in the tree from node A;
– either we keep the replica at node B and add one at node A or C, thereby having no traversing request any more.

The choice cannot be made locally, since it depends upon the remainder of the tree: if the root r has two client requests, then it was better to keep the pre-existing server B. However, if it has four requests, two

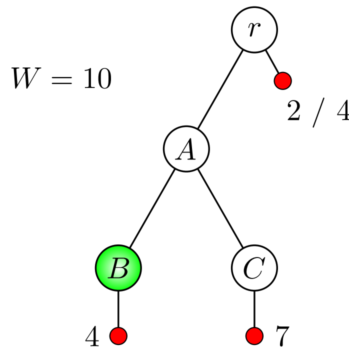


Figure 3.1: Example: reusing pre-existing replicas.

new servers are needed to satisfy all requests, and one can then remove server B which becomes useless (i.e., keep one server at node C and one server at node r).

From this example, it seems very difficult to design a greedy strategy to minimize the solution cost, while accounting for pre-existing replicas. We propose in the next section a dynamic programming algorithm which solves the MINCOST-WITHPRE problem.

3.3.2 Dynamic programming algorithm

Let W be the total number of requests that a server can handle, and r_i the number of requests issued by client $i \in \mathcal{C}$.

At each node $j \in \mathcal{N}$, we fill a table of maximum size $(E+1) \times (N-E+1)$ which indicates, for exactly $0 \leq e \leq E$ existing servers and $0 \leq n \leq N-E$ new servers in the subtree rooted in j (excluding j), the solution which leads to the minimum number of requests that have not been processed in the subtree. This solution for (e,n) values at node j is characterized by the minimum number of requests that is obtained, $minr^j_{(e,n)}$, and by the number of requests processed at each node $j' \in \mathsf{subtree}_j$, $req^j_{(e,n)}(j')$. Note that each entry of the table has a maximum size $O(N)$ (in particular, this size is reached at the root of the tree). The req variables ensure that it is possible to reconstruct the solution once the traversal of the tree is complete.

The call **init**(r) (see Algorithm 6), where r is the root of the tree, performs the initialization: tables are initialized to default values (no solution). We set $minr^j_{(e,n)} = W + 1$ to indicate that there is no solution, since in any valid solution, $minr^j_{(e,n)} \leq W$.

The main algorithm (see Algorithm 7) fills the tables while performing a bottom-up traversal of the tree, and the solution can be found within the table of the root node (see Algorithm 8, p. 62). Initially, we fill the table for nodes j which have only client nodes: $minr^j_{(0,0)} = \sum_{i \in \mathsf{children}_j \cap \mathcal{C}} r_i$, and $minr^j_{(k,l)} = W+1$ for $k>0$ or $l>0$. There are no nodes in the subtree of j, thus no req variables to set. The variable $client(j)$ keeps track of the number of requests directly issued by a client at node j. Also, recall that the decision whether to place a replica at node j or not is not accounted for in the table of j, but when processing the parent of node j.

Then, for a node $j \in \mathcal{N}$, we perform the same initialization, before processing children nodes one by one. The processing of child i of node j is done through the call to the **merge**(j,i) procedure (see Algorithm 9), and it is informally described below.

First, we copy the current table of node j into a temporary one, with values $tminr$ and $treq$. Note that the table is initially almost empty, but this copy is required since we process children one after the other, and when we call **merge**(j,i) for the kth children node, the table of j already contains information from the merge with the previous $k-1$ children nodes.

Then, for $0 \leq e \leq E$ and $0 \leq n \leq N-E$, we need to compute the new $minr^j_{(e,n)}$, and to update the $req^j_{(e,n)}$ values. We try all combinations with e' existing replicas and n' new replicas in the temporary table (i.e., information about children already processed), $e-e'$ existing replicas and $n-n'$ new replicas in the subtree of child i. We furthermore try solutions with a replica placed at node i, and we account for it in the value of e if $i \in \mathcal{E}$ (i.e., for a given value e', we place only $e-e'-1$ replica in the subtree of i, plus one on i); otherwise we account for it in the value of n. Each time we find a solution which is better than the one previously in the table (in terms of $minr$), we copy the values of req from the temporary table and the table of i, in order to retain all the information about the current best solution.

The key of the algorithm resides in the fact that during this *merging* process, the optimal solution will always be one which lets the minimum of requests pass through the subtree (see Lemma 3.1).

Algorithm 6: Initialization procedure.

```
procedure init (node j ∈ N)
for 0 ≤ e ≤ E do                                        /* Initializing the tables. */
    for 0 ≤ n ≤ N − E do minr^j_(e,n) = W + 1 ;          /* No solution. */
for i ∈ children_j ∩ N do init(i) ;                      /* Recursive call. */
```

Algorithm 7: Main procedure.

```
procedure main (node j ∈ N)
begin
    /* Init. client children. */
    client(j) = 0;
    for i ∈ children_j ∩ C do
        client(j) = client(j) + r_i;
    minr^j_(0,0) = client(j);
    if minr^j_(0,0) > W then exit(no solution);

    /* Processing child nodes. */
    for i ∈ children_j ∩ N do
        main(i); /* Recursive call. */
        merge(j, i);
end
end
```

The solution to the replica placement problem with pre-existing servers MINCOST-WITHPRE is computed through a call to **replica-update** (see Algorithm 8), which returns a set of replica $\mathcal{R}$ minimizing the cost: we scan all solutions in order to return a valid one of minimum cost.

To prove that the algorithm returns an optimal solution, we show in Lemma 3.1 that the solutions that are discarded while filling the tables, never lead to a better solution than the one that is finally returned.

Lemma 3.1. Consider a subtree rooted at node $j \in \mathcal{N}$. If an optimal solution uses e pre-existing servers and places n new servers in this subtree, then there exists an optimal solution of same cost, for which the placement of these servers minimizes the number of requests traversing j.

Proof. Let $\mathcal{R}_{opt}$ be the set of replicas in the optimal solution with (e, n) servers (i.e., e pre-existing and n new in $\mathsf{subtree}_j$). We denote by $rmin$ the minimum number of requests that must traverse j in a solution using (e, n) servers, and by $\mathcal{R}_{loc}$ the corresponding (local) placement of replicas in $\mathsf{subtree}_j$.

If $\mathcal{R}_{opt}$ is such that more than $rmin$ requests are traversing node j, we can build a new global solution which is similar to $\mathcal{R}_{opt}$, except for the subtree rooted in j for which we use the placement of $\mathcal{R}_{loc}$. The cost of the new solution is identical to the cost of $\mathcal{R}_{opt}$, therefore it is an optimal solution. It is still a valid solution, since $\mathcal{R}_{loc}$ is a valid solution and there are less requests than before to handle in the remaining of the tree (only $rmin$ requests traversing node j).

This proves that there exists an optimal solution which minimizes the number of requests traversing each node, given a number of pre-existing and new servers. ∎

The algorithm computes all local optimal solutions for all values (e, n). During the merge procedure, we try all possible numbers of pre-existing and new servers in each subtree, and we minimize the number of traversing requests, thus finding an optimal local solution. Thanks to Lemma 3.1, we know that there is a global optimal solution which builds upon these local optimal solutions.

Algorithm 8: Replica placement algorithm with pre-existing servers (MINCOST-WITHPRE problem).

algorithm **replica-update**
begin
- **init**(r);
- **main**(r);
- `/* Initially, no best solution. */`
- $cmin = N \times (1 + \text{create} + \text{delete})$;
- $minEN = (-1, -1)$;
- `/* Scanning root table: compute all costs for` (e, n). `*/`
- **for** $0 \le e \le E$ **do**
  - **for** $0 \le n \le N - E$ **do**
    - $req^r_{(e,n)}(r) = minr^r_{(e,n)}$;
    - $cost = N \times (1 + \text{create} + \text{delete})$;
    - **if** $minr^r_{(e,n)} = 0$ **then**
      - $cost = (e + n) + n \times \text{create} + (E - e) \times \text{delete}$;
    - **else if** $minr^r_{(e,n)} \le W$ *and* $r \in \mathcal{E}$ **then**
      - $cost = (e + n + 1) + n \times \text{create} + (E - e - 1) \times \text{delete}$;
    - **else if** $minr^r_{(e,n)} \le W$ *and* $r \in \mathcal{N} \setminus \mathcal{E}$ **then**
      - $cost = (e + n + 1) + (n + 1) \times \text{create} + (E - e) \times \text{delete}$;
    - `/* Check if this solution is better than previous best. */`
    - **if** $cost < cmin$ **then**
      - $cmin = cost$; $minEN = (e, n)$;
- `/* Reconstruct solution:` $\mathcal{R}$ `is the set of replicas. */`
- **if** $minEN = (-1, -1)$ **then exit**(*no solution*);
- **else**
  - $\mathcal{R} = \emptyset$;
  - **for** $j \in \mathcal{N}$ **do**
    - **if** $req^r_{minEN}(j) > 0$ **then** $\mathcal{R} = \mathcal{R} \cup \{j\}$;
  - **return**$(\mathcal{R})$;

end
end

Algorithm 9: Processing a child node.

procedure **merge**(j, i)

begin

- /* Duplicate table at node j, and clean up. */
- **for** $0 \leq e \leq E$ **do**
  - **for** $0 \leq n \leq N - E$ **do**
    - $tminr_{(e,n)} = minr^j_{(e,n)}$;
    - $minr^j_{(e,n)} = W + 1$; /* No solution in the merged table. */
    - **for** $j' \in$ *subtree*$_j \cap \mathcal{N}$ **do** $treq_{(e,n)}(j') = req^j_{(e,n)}(j')$;
- /* Try all solutions with e existing replicas and n new replicas. */
- **for** $0 \leq e \leq E$ **do** **for** $0 \leq n \leq N - E$ **do**
  - **for** $0 \leq e' \leq e$ **do** **for** $0 \leq n' \leq n$ **do**
    - **if** $tminr_{(e',n')} \leq W$ **then**
      - /* e' existing and n' new on children already processed, $e - e'$ existing and $n - n'$ new in the subtree of i, no replica on i. */
      - **if** $minr^i_{(e-e',n-n')} + tminr_{(e',n')} \leq \min(W, minr^j_{(e,n)})$ **then**
        - /* Better solution than existing one for (e,n). */
        - $minr^j_{(e,n)} = minr^i_{(e-e',n-n')} + tminr_{(e',n')}$;
        - **for** $j' \in$ *subtree*$_j \cap \mathcal{N}$ **do**
          - **if** $j' \in$ *subtree*$_i$ **then** $req^j_{(e,n)}(j') = req^i_{(e-e',n-n')}(j')$;
          - **else** $req^j_{(e,n)}(j') = treq_{(e',n')}(j')$;
      - /* e' existing and n' new on children already processed, replica on i. */
      - **if** $(i \in \mathcal{E})$ *and* $(e' < e)$ **then**
        - /* $e - e' - 1$ existing and $n - n'$ new in the subtree of i. */
        - **if** $tminr_{(e',n')} \leq minr^j_{(e,n)}$ **then**
          - /* Better solution than existing one for (e,n). */
          - $minr^j_{(e,n)} = tminr_{(e',n')}$;
          - **for** $j' \in$ *subtree*$_j \cap \mathcal{N}$ **do**
            - **if** $j' \in$ *subtree*$_i$ **then** $req^j_{(e,n)}(j') = req^i_{(e-e'-1,n-n')}(j')$;
            - **else** $req^j_{(e,n)}(j') = treq_{(e',n')}(j')$;
          - $req^j_{(e,n)}(i) = minr^i_{(e-e'-1,n-n')}$;
      - **else if** $(i \notin \mathcal{E})$ *and* $(n' < n)$ **then**
        - /* $e - e'$ existing and $n - n' - 1$ new in the subtree of i. */
        - **if** $tminr_{(e',n')} \leq minr^j_{(e,n)}$ **then**
          - /* Better solution than existing one for (e,n). */
          - $minr^j_{(e,n)} = tminr_{(e',n')}$;
          - **for** $j' \in$ *subtree*$_j \cap \mathcal{N}$ **do**
            - **if** $j' \in$ *subtree*$_i$ **then** $req^j_{(e,n)}(j') = req^i_{(e-e',n-n'-1)}(j')$;
            - **else** $req^j_{(e,n)}(j') = treq_{(e',n')}(j')$;
          - $req^j_{(e,n)}(i) = minr^i_{(e-e',n-n'-1)}$;

end

end

3.3.3 Execution time of the algorithm

Recall that N is the total number of nodes, and E is the number of pre-existing nodes.

The call to **init**(r) makes a traversal of the tree, and at each node, the table of size $O((E+1) \times (N-E+1))$ is initialized. The total cost for this call is therefore in $O(N \times (N-E+1) \times (E+1))$.

For the main procedure, the processing of a node with only client children is done in constant time $O(1)$. The processing of each non-client child consists in a call to the **merge** procedure, and there is only one such per node of the tree, and therefore N calls to this procedure during the whole execution.

The initialization of the merging procedure takes a time $O((N-E+1) \times (E+1))$. Then, we try all solutions with $e \leq E$ existing replicas, and $n \leq N-E$ new replicas. Given e and n, there are no more than $O((N-E+1) \times (E+1))$ possible solutions ($0 \leq e' \leq e \leq E$ existing replicas and $0 \leq n' \leq n \leq N-E$ new replicas on the children already processed, with or without a replica on the child currently being processed). Finally, the total number of iterations in the loop is bounded by $O((N-E+1)^2 \times (E+1)^2)$. The most consuming operation in the loop is to copy the req variables, which is done in $O(N)$. However, this copy can be done outside the loop: we keep track of the best solution for each couple (e, n), and update the req variables in another loop over (e, n). It is done by decreasing values of e and n, since the update for (e, n) requires the non-updated values with (e', n') such that $e' \leq e$ and $n' \leq n$. The total cost with this optimization (see [100] for the implementation) is therefore the number of iterations, i.e., $O((N-E+1)^2 \times (E+1)^2)$.

Then, scanning the table at the root is done in $O((N-E+1) \times (E+1))$, and reconstructing the solution takes a single tree traversal, i.e., it is in $O(N)$. Finally, the complexity of the dynamic closest replica placement algorithm is in $O(N \times (N-E+1)^2 \times (E+1)^2)$, which corresponds to the N calls to the merging procedure. The algorithm is therefore of polynomial complexity, at most $O(N^5)$ for a tree with N nodes. This concludes the proof of Theorem 3.1.

3.4 Complexity results with power

In this section, we tackle the MINPOWER and MINPOWER-BOUNDEDCOST problems. First in Section 3.4.1, we use an example to show why minimizing the number of requests traversing the root of a subtree is no longer optimal, and we illustrate the difficulty to take local decisions even when restricting to the simpler mono-criterion MINPOWER problem. Then in Section 3.4.2, we prove the NP-completeness of the latter problem with an arbitrary number of modes (Theorem 3.2). However, we propose a pseudo-polynomial algorithm to solve the problem in Section 3.4.3. This algorithm turns out to be polynomial when the number of modes is constant, hence usable in a realistic setting with two or three modes (Theorem 3.3).

3.4.1 Running example

Consider the example of Figure 3.2. There are two modes, $W_1 = 7$ and $W_2 = 10$, and we focus on the power minimization problem. For simplicity, we assume that the power consumption of a node running at mode W_i is $400 + W_i^3$, for $i = 1, 2$ (400 is the static power, and we set $\alpha = 3$). We consider the subtree rooted in A. Several decisions can be taken locally:

- place a server at node A, running at mode W_2, hence minimizing the number of traversing requests. Another solution without traversing requests is to have two servers, one at node B and one at node C, both running at mode W_1, but this would lead to a higher power consumption, since $800 + 2 \times 7^3 > 400 + 10^3$;
- place a server running at mode W_1 at node C, thus having 3 requests going through node A.

The choice cannot be made greedily, since it depends upon the rest of the tree: if the root r has four client requests, then it is better to let some requests through (one server at node C), since it optimizes power consumption. However, if it has ten requests, it is necessary to have no request going through A, otherwise node r is not able to process all its requests.

From this example, it seems very hard to design a greedy strategy to minimize the power consumption. Similarly, if we would like to reuse the algorithm of Section 3.3 to solve the MINPOWER-BOUNDEDCOST-WITHPRE bi-criteria problem, we would need to account for modes. Indeed, the best solution of subtree A with one server is no longer always the one which minimizes the number of requests (in this case, placing one server on node A), since it can be better for power consumption to let three requests traverse node A and balance the load upper in the tree.

We prove in the next section the NP-completeness of the problem, when the number of modes is arbitrary. However, we can adapt the dynamic programming algorithm, which becomes exponential in the number of modes, but hence remains polynomial for a constant number of modes (see Section 3.4.3).

3.4.2 NP-completeness of MINPOWER

In this section, we prove Theorem 3.2, i.e., the NP-completeness of the MINPOWER problem, even with no static power, when there is an arbitrary number of modes.

Proof of Theorem 3.2. We consider the associated decision problem: given a total power consumption $\mathcal{P}$, is there a solution which does not consume more than $\mathcal{P}$?

First, the problem is clearly in NP: given a solution, i.e., a set of servers, and the mode of each server, it is easy to check in polynomial time that no capacity constraint is exceeded, and that the power consumption meets the bound.

To establish the completeness, we use a reduction from 2-Partition [44]. We consider an instance $\mathcal{I}_1$ of 2-Partition: given n strictly positive integers $a_1, a_2, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$; we assume that S is even (otherwise there is no solution).

We build an instance $\mathcal{I}_2$ of our problem where each server has $n+2$ modes. We assume that the a_i are sorted in increasing order, i.e., $a_1 \leq \cdots \leq a_n$. The modes are then, in increasing order:

- $W_1 = K$;
- $\forall 1 \leq i \leq n, W_{i+1} = K + a_i \times X$;
- $W_{n+2} = K + S \times X$;

where the values of K and X will be determined later.

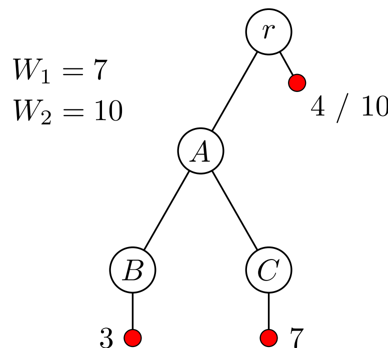


Figure 3.2: Example: minimizing power consumption.

We furthermore set that there is no static power, and the power consumption for a server running at capacity W_i is therefore $\mathcal{P}_i = W_i^\alpha$, where α is the rational exponent used in the computation of the power (see Section 3.2), and $2 \leq \alpha \leq 3$. The idea is to have K large and X small, so that we have an upper bound on the power consumed by a server running at capacity W_{i+1}, for $1 \leq i \leq n$:

$$W_{i+1}^\alpha = (K + a_i \times X)^\alpha \leq K^\alpha + a_i + \frac{1}{n}. \tag{3.5}$$

To ensure that Equation (3.5) is satisfied, we set

$$X = \frac{1}{\alpha \times K^{\alpha-1}},$$

and then we have $(K + a_i \times X)^\alpha = K^\alpha(1 + \frac{a_i}{\alpha K^\alpha})^\alpha$, with $K > S$ and therefore $\frac{a_i}{\alpha K^\alpha} < 1$. We set $x_i = \frac{a_i}{\alpha K^\alpha}$, and we want to ensure that:

$$(1 + x_i)^\alpha \leq 1 + \alpha \times x_i + \frac{1}{n \times K^\alpha}. \tag{3.6}$$

To do so, we study the function

$$f(x) = (1 + x)^\alpha - (1 + \alpha \times x) - 5x^2,$$

and we show that $f(x) \leq 0$ for $x \leq \frac{1}{2}$ (thanks to the term in $-5x^2$).

We have $f(0) = 0$, and $f'(x) = \alpha(1 + x)^{\alpha-1} - \alpha - 10x$. We have $f'(0) = 0$, and $f''(x) = \alpha(\alpha - 1)(1 + x)^{\alpha-2} - 10$. Since $\alpha \leq 3$, $\alpha(\alpha - 1)(1 + x)^{\alpha-2} \leq 6(1 + x)$, and for $x \leq \frac{1}{2}$, $f''(x) < 0$. We deduce that $f'(x)$ is non increasing for $x \leq \frac{1}{2}$, and since $f'(0) = 0$, $f'(x)$ is negative for $x \leq \frac{1}{2}$. Finally, $f(x)$ is non increasing for $x \leq \frac{1}{2}$, and since $f(0) = 0$, we have $(1 + x)^\alpha < (1 + \alpha \times x) + 5x^2$ for $x \leq \frac{1}{2}$.

Equation (3.6) is therefore satisfied if $5x_i^2 \leq \frac{1}{n \times K^\alpha}$, i.e., $K^\alpha \geq \frac{5a_i^2 \times n}{\alpha^2}$. This condition is satisfied for

$$K = n \times S^2,$$

and we then have $x_i < \frac{1}{2}$, which ensures that the previous reasoning was correct. Finally, with these values of K and X, Equation (3.5) is satisfied.

Then, the distribution tree is the following: the root node r has one client with $K + \frac{S}{2} \times X$ requests, and n children $A_1, \dots, A_n$. Each node A_i has a client with $a_i \times X$ requests, and a children node B_i which has K requests. Figure 3.3 illustrates the instance of the reduction.

Finally, we ask if we can find a placement of replicas with a maximum power consumption of:

$$\mathcal{P}_{max} = (K + S \times X)^\alpha + n \times K^\alpha + \frac{S}{2} + \frac{n-1}{n}.$$

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$, since K and X are of polynomial size. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Let us assume first that $\mathcal{I}_1$ has a solution, I. The solution for $\mathcal{I}_2$ is then as follows: there is one server at the root, running at capacity W_{n+2}. Then, for $i \in I$, we place a server at node A_i running at capacity W_{1+i}, while for $i \notin I$, we place a server at node B_i running at capacity W_1. It is easy to check that all capacity constraints are satisfied for nodes A_i and B_i. At the root of the tree, there are $K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X$, which sums up to $K + S \times X$. The total power consumption is then $\mathcal{P} = (K + S \times X)^\alpha + \sum_{i \in I}(K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha$. Thanks to Equation (3.5), $\mathcal{P} \leq (K + S \times$

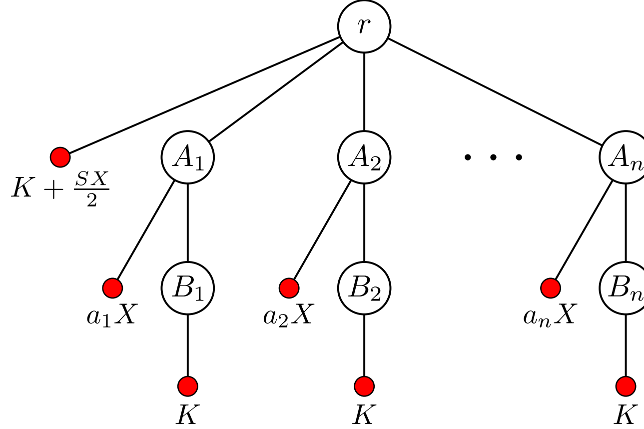


Figure 3.3: Illustration of the NP-completeness proof.

$X)^\alpha + \sum_{i \in I} \left(K^\alpha + a_i + \frac{1}{n}\right) + \sum_{i \notin I} K^\alpha$, and finally, $\mathcal{P} \leq (K + S \times X)^\alpha + n \times K^\alpha + \sum_{i \in I} a_i + \frac{n-1}{n}$. Since I is a solution to 2-Partition, we have $\mathcal{P} \leq \mathcal{P}_{max}$. Finally, $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. There is a server at the root node r, which runs at mode W_{n+2}, since this is the only way to handle its $K + \frac{S}{2} \times X$ requests. This server has a power consumption of $(K + S \times X)^\alpha$. Then, there cannot be more than n other servers. Indeed, if there were $n + 1$ servers, running at the smallest mode W_1, their power consumption would be $(n+1)K^\alpha$, which is strictly greater than $n \times K^\alpha + \frac{S}{2} + 1$. Therefore, the power consumption would exceed $\mathcal{P}_{max}$. So, there are at most n extra servers.

Consider that there exists $i \in \{1, \dots, n\}$ such that there is no server, neither on A_i nor on B_i. Then, the number of requests at node r is at least $2K$; however, $2K > W_{n+2}$, so the server cannot handle all these requests. Therefore, for each $i \in \{1, \dots, n\}$, there is exactly one server either on A_i or on B_i. We define the set I as the indices for which there is a server at node A_i in the solution. Now we show that I is a solution to $\mathcal{I}_1$, the original instance of 2-Partition.

First, if we sum up the requests at the root node, we have:

$$K + \frac{S}{2} \times X + \sum_{i \notin I} a_i \times X \leq K + S \times X.$$

Therefore, $\sum_{i \notin I} a_i \leq \frac{S}{2}$.

Now, if we consider the power consumption of the solution, we have:

$$(K + S \times X)^\alpha + \sum_{i \in I} (K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha \leq \mathcal{P}_{max}.$$

Let us assume that $\sum_{i \in I} a_i > \frac{S}{2}$. Since the a_i are integers, we have $\sum_{i \in I} a_i \geq \frac{S}{2} + 1$. It is easy to see that $(K + a_i \times X)^\alpha > K^\alpha + a_i$. Finally, $\sum_{i \in I} (K + a_i \times X)^\alpha + \sum_{i \notin I} K^\alpha \geq n \times K^\alpha + \sum_{i \in I} a_i \geq n \times K^\alpha + \frac{S}{2} + 1$. This implies that the total power consumption is greater than $\mathcal{P}_{max}$, which leads to a contradiction, and therefore $\sum_{i \in I} a_i \leq \frac{S}{2}$.

We conclude that $\sum_{i \notin I} a_i = \sum_{i \in I} a_i = \frac{S}{2}$, and so the solution I is a 2-Partition for instance $\mathcal{I}_1$. This concludes the proof. ■

3.4.3 A pseudo-polynomial algorithm for MINPOWER-BOUNDEDCOST

In this section, we sketch how to adapt the algorithm of Section 3.3 to account for power consumption. As illustrated in the example of Section 3.4.1, the current algorithm may lead to a non-optimal solution for the power consumption if used only with the higher mode for servers. Therefore, we refine it and compute, in each subtree, the optimal solution with, for $1 \leq j, j' \leq M$,

– exactly n_j new servers running at mode W_j;
– exactly $e_{j,j'}$ pre-existing servers whose operation modes have changed from W_j to $W_{j'}$.

Recall that we previously had only two parameters, n the number of new servers, and e the number of pre-existing servers, thus leading to a total of $(N - E + 1)^2 \times (E + 1)^2$ iterations for the **merge** procedure (Lines 8-9 of Algorithm 9). Now, the number of iterations is $(N - E + 1)^{2M} \times (E + 1)^{2M^2}$, since we have $2 \times M$ loops of maximum size $N - E + 1$ over the n_j and n'_j, and $2 \times M^2$ loops of maximum size $E + 1$ over the $e_{j,j'}$ and $e'_{j,j'}$.

The new algorithm is similar, except that during the merge procedure, we must consider the type of the current node that we are processing (existing or not), and furthermore set it to all possible modes. This is done at Lines 16 and 23 of Algorithm 9, when we try to add a server at node i. We therefore add a loop of size M.

We do not formalize the new **merge** procedure, since its principle is similar to Algorithm 9, except that we need to have larger tables at each node, and to iterate over all parameters. The complexity of the N calls to this procedure is now in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

Of course, we need also to update the **init** and **main** procedures to account for the increasing number of parameters. Finally, we rewrite the equivalent of Algorithm 8 but according to the bi-criteria objective function: first we compute all costs, accounting for the cost of changing modes, and then we scan all solutions, and return one whose cost is not greater than the threshold, and which minimizes the power consumption. The most time-consuming part of the algorithm is still the call to the **merge** procedures, hence a complexity in $O(N \times M \times (N - E + 1)^{2M} \times (E + 1)^{2M^2})$.

With a constant number of capacities, this algorithm is polynomial, which proves Theorem 3.3. For instance, with $M = 2$, the worst case complexity is $O(N^{13})$. Without pre-existing servers, this complexity is reduced to $O(N^5)$.

3.5 Simulations

In this section, we compare our algorithms with the algorithms of [121], which do not account for pre-existing servers and for power consumption. First in Section 3.5.1, we focus on the impact of pre-existing servers. Then we consider the power consumption minimization criterion in Section 3.5.2.

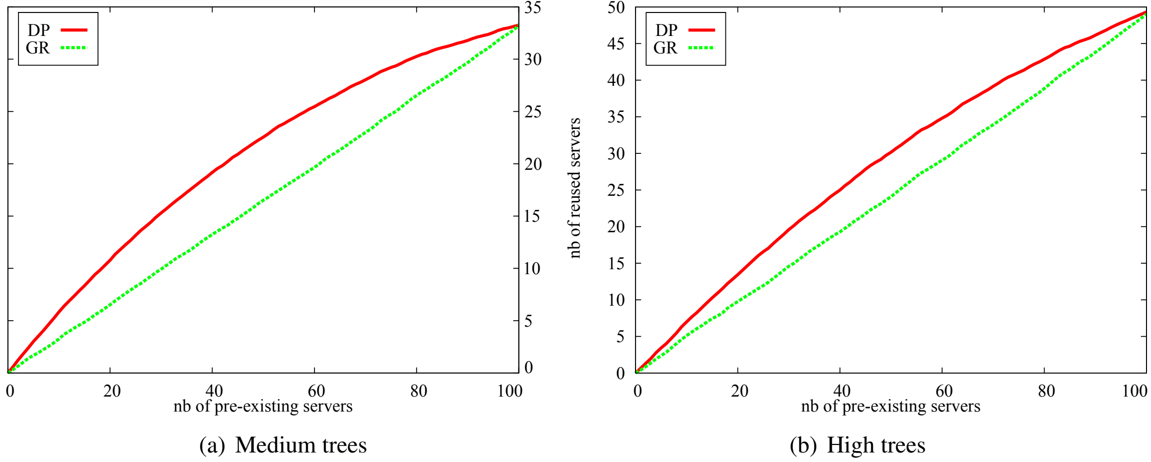
Note that experiments have been run sequentially on an Intel Xeon 5250 processor, and run sequentially. The source code of all algorithms and simulations is publicly available on the Web [100].

3.5.1 Impact of pre-existing servers

In this set of experiments, we randomly build a set of distribution trees with $N = 100$ internal nodes of maximum capacity $W = 10$. Each internal node has between 6 and 9 children, and clients are distributed randomly throughout the tree: each internal node has a client with a probability 0.5, and this client has between 1 and 6 requests.

In the first experiment, we draw 200 random trees without any existing replica in them. Then we randomly add $0 \leq E \leq 100$ pre-existing servers in each tree. Finally, we execute both the greedy algorithm (GR) of [121], and the algorithm of Section 3.3 (DP) on each tree, and since both algorithms

return a solution with the minimum number of replicas, the cost of the solution is directly related to the number of pre-existing replicas that are reused. Figure 3.4(a) shows the average number of pre-existing servers that are reused in each solution over the 200 trees, for each value of the number E of pre-existing servers. When the tree has a very small ($E \approx 0$) or very large ($E \approx N$) number of pre-existing replicas, both algorithms return the same solution. Still, DP achieves an average reuse of 4.13 more servers than GR, and it can reuse up to 15 more servers.



(a) Medium trees (b) High trees

Figure 3.4: Experiment 1: increasing number of pre-existing servers.

In a second experiment, we study the behavior of the algorithms in a *dynamic* setting, with 20 update steps. At each step, starting from the current solution, we update the number of requests per client and recompute an optimal solution with both algorithms, starting from the servers that were placed at the previous step. Initially, there are no pre-existing servers, and at each step, both algorithms obtain a different solution. However, they always reach the same total number of servers since they have the same requests; but after the first step, they may have a different set of pre-existing servers. Similarly to Experiment 1, the simulation is conducted on 200 distinct trees, and results are averaged over all trees.

In Figure 3.5(a) (left), at each step, we compare the number of existing replicas in the solutions found by the two algorithms, and hence the cost of the solutions. We plot the cumulative number of servers that have been reused so far (hence accounting for all previous steps). The DP algorithm makes a better reuse of pre-existing replicas. Figure 3.5(a) (right) compares, at each step, the number of pre-existing servers reused by DP and by GR. We count the average number of steps (over 20) at which each value is reached. It occasionally happens that the greedy algorithm performs a better reuse, because it is not starting from the same set of pre-existing servers, but overall this experiment confirms the better reuse of the dynamic programming algorithm, even when the algorithms are applied on successive steps.

Note however that taking pre-existing replicas into account has an impact on the execution time of the algorithm: in these experiments, GR runs in less than one second per tree, while DP takes around forty seconds per tree.

Also, we point out that the shape of the trees does not seem to modify the general behaviour: the results with trees where each node has between 2 and 4 children are depicted in Figure 3.4(b) and Figure 3.5(b).

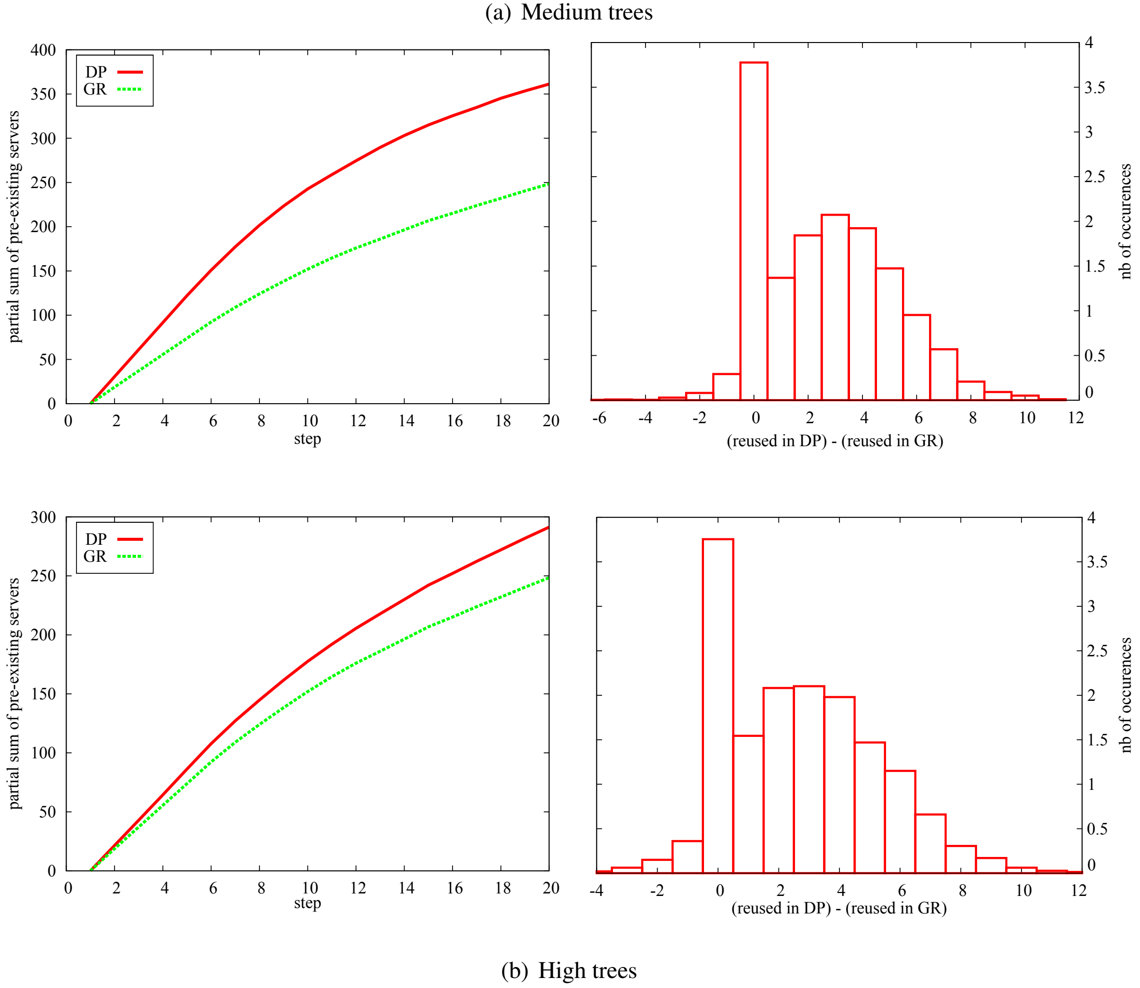


Figure 3.5: Experiment 2: consecutive executions of the algorithms.

3.5.2 With power consumption

To study the practical applicability of the bi-criteria algorithm (DP) for the MINPOWER-BOUNDEDCOST problem (see Section 3.4.3), we have implemented it with two modes $W_1 = 5$ and $W_2 = 10$, and compared it with the algorithm in [121]; this algorithm does not account for power minimization, but minimizes the value of the maximal capacity W when given a cost bound. More precisely, in the experiment we try all values $5 \leq W \leq 10$, and compute the corresponding cost and power consumption. To be fair, when a server has 5 requests or less, we operate it under the first mode W_1. Given a bound on the cost, we keep the solution that minimizes the power consumption. We call GR this version of the algorithm in [121] modified for power as explained above.

We randomly build 100 trees with 50 nodes each, and we select 5 nodes as pre-existing servers. Clients have between 1 and 5 requests, so that a solution with replicas in the first mode can always be found. The cost function is such that, for any $i, i' \in \{1, 2\}$, $\mathsf{create}_i = 0.1$, $\mathsf{delete}_i = 0.01$ and $\mathsf{changed}_{i,i'} = 0.001$. The power consumed by a server in mode i is $\mathcal{P}_i = \frac{1}{10} W_1^3 + W_i^3$. In Figure 3.6(a), we plot the inverse of the power of a solution, given a bound on the cost (the higher the better). If the algorithm fails to find a solution for a tree, the value is 0, and we average the inverse of the power over

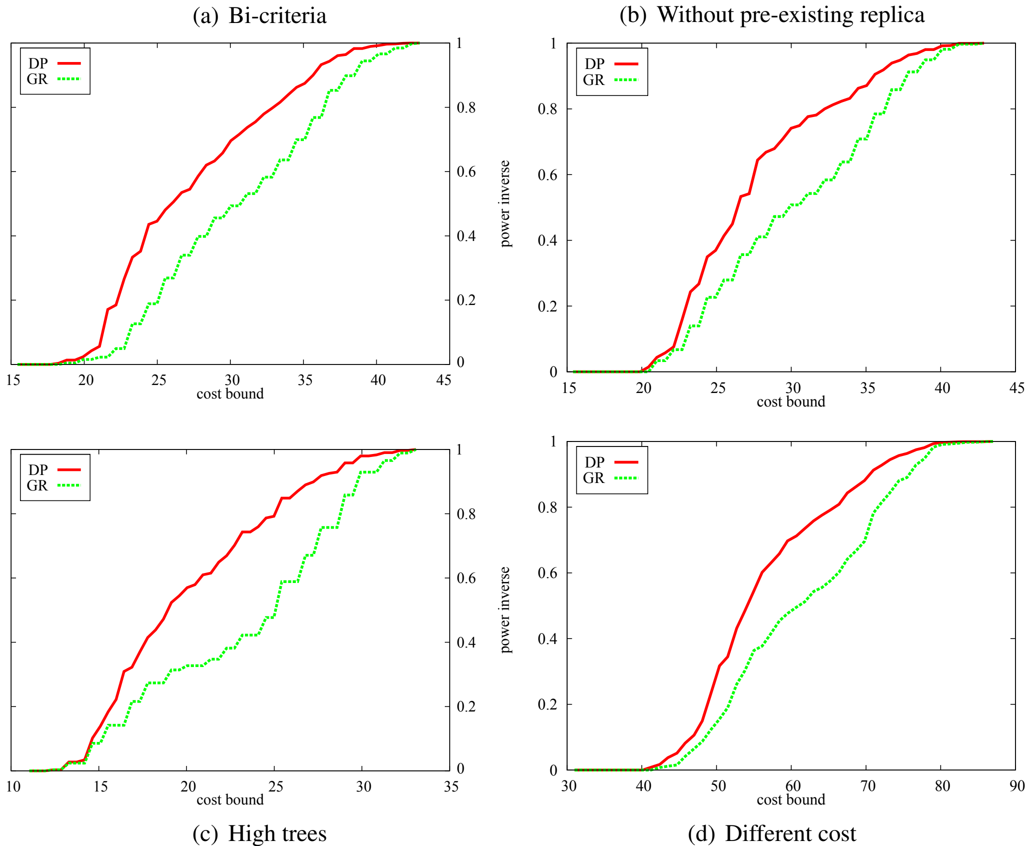
the 100 trees, for both algorithms. For intermediate cost values, our algorithm is much better than the version of [121] in terms of power consumption: GR consumes in average more than 30% more power than DP, when the cost bound is between 29 and 34.

Here again, it takes more time to obtain the optimal solution with DP than to run the greedy algorithm several times: GR runs in around $1\,s$ per tree, while DP takes around $5\,min$ per tree. Also, we have performed some more experiments with slightly different parameters, and got only little differences.

First, we look at the power part of the DP algorithm, running on trees without pre-existing replicas (see Figure 3.6(b)). For low bound costs, the two curves are close, because DP finds a solution if and only if GR finds one, and the dissipated power is high; there is no significant difference for other costs.

Then, we run the experiment on high trees (each internal node has from 2 to 4 children). Results are shown in Figure 3.6(c). The ratio between the dynamic programming algorithm and the greedy one is better than the ratio on fat trees for intermediate costs: when the bound cost is between 22 and 27, GR consumes in average more than 40% more power than DP, and 60% between 23 and 25.

Finally, in Figure 3.6(d), we show the results for a cost function such that deleting and creating costs are high (more precisely, for any $i, i' \in \{1, 2\}$, $\mathsf{create}_i = \mathsf{delete}_i = 1$ and $\mathsf{changed}_{i,i'} = 0.1$). Compared to the initial costs, the ratio between DP and GR is greater for lowest cost bounds, because GR finds less solutions than DP. Indeed, DP can find solutions with lower cost, taking pre-existing replicas into account.

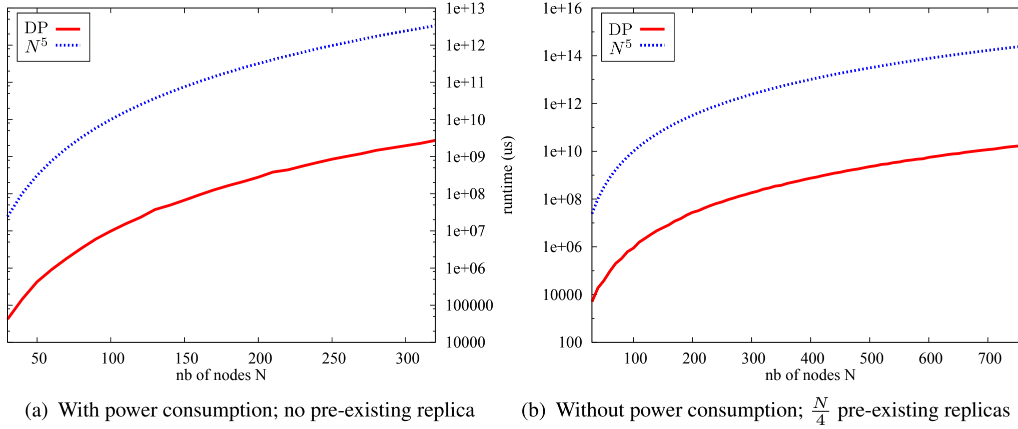


(c) High trees
(d) Different cost

Figure 3.6: Experiment 3.

3.5.3 Running time of the algorithms

Recall that the theoretical complexity of GR is of order $O(N \log N)$ (without power and without pre-existing servers), while DP is of order $O(N^5)$, both for the version with power (two modes) but without pre-existing servers, and for the version without power but with pre-existing servers. In practice, the run times of GR are very small (a few milliseconds). For DP, we have plotted its run time as a function of N, see Figure 3.7. Run time measurements show that the experimental values have a shape in N^5, which confirms the theoretical complexity. Moreover, our DP algorithms run in less than N^5 microseconds for reasonable values of N, which allows the use of these algorithms in practical situations.



(a) With power consumption; no pre-existing replica (b) Without power consumption; $\frac{N}{4}$ pre-existing replicas

Figure 3.7: Running time of the DP algorithm.

Indeed, without power, we are able to process trees with 500 nodes and 125 pre-existing servers in 30 minutes; with power and no pre-existing server, we can process trees with 300 nodes in one hour. The algorithm with power and pre-existing servers is the most time-consuming: it takes around one hour to process a tree with 70 nodes and 10 pre-existing servers.

3.6 Conclusion

In this chapter, we have addressed the problem of updating the placement of replicas in a tree network. We have provided an optimal dynamic programming algorithm whose cost is at most $O(N^5)$, where N is the number of nodes in the tree. This complexity may seem high for very large problem sizes, but our implementation of the algorithm is capable of managing trees with up to 500 nodes in half an hour, which is reasonable for a large spectrum of applications (e.g., such as database updates during the night).

The optimal placement update algorithm is a first step towards dealing with dynamic replica management. When client requests evolve over time, the placement of the replicas must be updated at regular intervals, and the overall cost is a trade-off between two extreme strategies: (i) "lazy" updates, where there is an update only when the current placement is no longer valid; the update cost is minimized, but changes in request volume and location since the last placement may well lead to poor resource usage; and (ii) systematic updates, where there is an update every time-step; this leads to an optimized resource usage but encompasses a high update cost. Clearly, the rates and amplitudes of the variations of the number of requests issued by each client in the tree are very important to decide for a good update interval. Still, establishing the cost of an update is a key result to guide such a decision. When un-frequent

updates are called for, or when resources have a high cost, the best solution is likely to use our optimal but expensive algorithm. On the contrary, with frequent updates or low-cost servers, we may prefer to resort to faster (but sub-optimal) update heuristics.

Our main contribution is to have provided the theoretical foundations for a single step reconfiguration, whose complexity is important to guide the design of lower-cost heuristics. Also, we have done a first attempt to take power consumption into account, in addition to usual performance-related objectives. Power consumption has become a very important concern, both for economic and environmental reasons, and it is important to account for it when designing replica placement strategies.

Even though our optimal algorithms have a high worst-case complexity, we have successfully implemented all of them, including the most time-consuming scheme capable of optimizing power while enforcing a bounded cost that includes pre-existing servers. We were able to process trees with a reasonable number of nodes.

As future work, we plan to design polynomial time heuristics with a lower complexity than the optimal solution. The idea would be to perform some local optimizations to better load-balance the number of requests per replica, with the goal of minimizing the power consumption. These heuristics should be tuned for dedicated applications, and should (hopefully!) build upon the fundamental results (complexity and algorithms) that we have provided in this chapter. Finally, it would be interesting to add more parameters in the model, such as the cost of routing, or the introduction of quality of service constraints.

Chapter 4

Mapping series-parallel workflows onto chip multiprocessors

4.1 Introduction

In the previous chapters, the elements (tasks or requests) that had to be computed were successively independent, distributed on chains of dependencies and finally placed on the leaves of a tree. In this chapter, we aim at minimizing the energy consumption of streaming applications whose task graph is a series-parallel graph (SPG). As said in Chapter 2, streaming applications, or workflows, are ubiquitous in many domains, as for instance image processing applications, astrophysics, meteorology, neuroscience, and so on [39, 108, 103, 129]. Most of these applications have simple and regular task graphs, such as linear chains (see Chapter 2), trees, fork-join graphs, or general SPGs (see Section 4.3.1 for a formal definition of SPGs). For instance, all the benchmarks of the *StreamIt* suite [109] are SPGs.

The performance-related objective coupled with energy minimization is the *period* of the streaming application. We recall that a series of data sets enter the input stage and progress from stage to stage, following the dependencies of the application, until the final result is computed. Each stage has its own communication and computation requirements: it reads inputs from the previous stage(s), processes the data and outputs results to the next stage(s). The pipeline operates in a dataflow mode: after a transient behavior due to the initialization delay, a new data set is completed every period. The period, which corresponds to the inverse of the throughput, is a key performance-related objective for streaming applications [110, 39, 51]. Formally, the period is the time interval between the arrival of two consecutive data sets in the application. Given a mapping of the application onto a platform, the time spent in each resource (processor or communication link) should not exceed the period.

Finally, the target platform for this study is a Chip MultiProcessor (CMP), which is composed of $p \times q$ homogeneous cores arranged along a 2D grid. During the last century, advances in integrated circuit technology have led chip designers to increase microprocessor performance by increasing the integration density thus allowing for higher clock rates and new innovations in micro-architectures. Such innovations included wider instructions, speculative execution, branch prediction and dynamic scheduling. However, in 1996, Olukotun et al. [91] argued that such a trend would not continue because of the diminishing return caused by limited instruction level parallelism and they argued that a better way for using the denser integration would be to layout multiple simpler processors on the same chip. Moreover, power consumption consideration prevented the push towards faster clocks, thus leaving the design of chip multiprocessors as the only alternative for increasing the on-chip computational capability. Specifically, increasing the number of cores rather than the processor's complexity translates into slower growth in power consumption. Currently, chip multiprocessors are commercially available and the challenge is

now to be able to efficiently utilize the parallelism available on chip.

An essential step for exploring the parallelism available in a streaming application is to provide algorithms and scheduling strategies for mapping a series-parallel graph onto a CMP, with the objective of minimizing the energy consumption while not exceeding a prescribed period. In some applications, data sets arrive at fixed time intervals, and hence the period of the application is given a priori, before any mapping is computed. In other applications, there is the freedom to choose between a set of possible periods, which are prescribed by the user. In all cases, the main goal is to reduce the energy consumption of the mapping, while enforcing the constraint on the prescribed period.

The contribution of this chapter is twofold. On the theoretical side, we assess the complexity of the above-mentioned mapping problem, using a *DAG-partition* mapping rule that partitions the application SPG into an acyclic graph of node clusters. In turn, each cluster is mapped onto a different processor of the CMP. Our cost model accounts for communication delays and cost (in terms of consumed energy). The problem turns out to be NP-hard, so we also study the complexity of simpler problem instances, either with a simpler target platform (uni-directional or bi-directional uni-line CMP), or by restricting to particular applications whose graph has a bounded degree of parallelism (*bounded-elevation* SPGs). The only problem instance that can be solved in polynomial time, thanks to a dynamic programming algorithm, is the mapping of bounded-elevation SPGs onto a uni-directional uni-line CMP. For other problem instances, we provide sophisticated NP-completeness proofs. On the practical side, building upon the theoretical results, we design some polynomial-time heuristics to solve the most general problem, and we assess their performance through simulations.

The chapter is organized as follows. We first survey related work in Section 4.2. Then we detail the framework in Section 4.3, and we provide complexity results in Section 4.4. The heuristics are described in Section 4.5, and simulation results in Section 4.6. Finally, we conclude and discuss future research directions in Section 4.7.

4.2 Related work

Reducing the energy consumption of computational platforms is an important research topic, and many techniques at the process, circuit design, and micro-architectural levels have been proposed [76, 74, 49]. The dynamic voltage and frequency scaling (DVFS) technique has been extensively studied, since it may lead to efficient energy/performance trade-offs [65, 45, 13, 29, 70, 127, 120]. Current microprocessors (for instance, from AMD [5] and Intel [88]) allow the speed to be set dynamically. Indeed, by lowering supply voltage, hence processor clock frequency, it is possible to achieve important reductions in power consumption, without necessarily increasing the execution time.

In this chapter, we aim at minimizing the energy consumption for series-parallel graph (SPG) applications which are mapped onto a chip multiprocessor (CMP). We first discuss related work on SPG applications, then we review different energy minimization approaches. Finally, we relate work on mapping problems on CMPs.

Series-parallel workflow applications. Classical workflow applications usually consists of a directed acyclic graph: the application is made of several tasks, and there are dependencies between these tasks. However, it turns out that many of these graphs are series-parallel graphs. For instance, in [84], McClatchey et al. introduce a prototype scientific workflow management system entitled CRISTAL, and the distributed scientific workflow applications that they consider are SPGs. In [96], Qin and Fahringer discuss several scientific grid workflow applications, which are all structured as SPGs: the WIEN2k workflow performs electronic structure calculations of solids using density functional theory [20], the MeteoAG workflow is a meteorology simulation application [103], and the GRASIL workflow calculates

the spectral energy distribution of galaxies [108]; this latter application has actually a fork-join graph. A last example is the fMRI workflow [129], which is a cognitive neuroscience application.

DVFS and optimization problems. Part of this related work had already been invoked in the Chapter 2. In [90], Okuma et al. demonstrate that voltage scaling is far more effective than the shutdown approach, which simply stops the power supply when the system is inactive. Chen et al. [27] consider parallel sparse applications, and they show that when scheduling applications modeled by a directed acyclic graph with a well-identified critical path, it is possible to lower the voltage during non-critical execution of tasks, with no impact on the execution time. Similarly, Wang et al. [120] study the slack time for non-critical jobs, they extend their execution time and thus reduce the energy consumption without increasing the total execution time. Kim et al. [70] provide power-aware scheduling algorithms for bag-of-tasks applications with deadline constraints, based on dynamic voltage scaling. Their goal is to minimize power consumption as well as to meet the deadlines specified by application users. In the context of real-time embedded systems, Lee and Sakurai [76] show how to exploit slack time arising from workload variation, thanks to a software feedback control of supply voltage. Prathipati [94] discusses techniques to take advantage of run-time variations in the execution time of tasks; it determines the minimum voltage under which each task can be executed, while guaranteeing the deadlines of each task. In [126], dynamic programming algorithms are given to minimize the expected energy consumption in real time systems using frequency and voltage scaling. Yang and Lin [127] discuss algorithms with preemption, using DVS techniques; substantial energy can be saved using these algorithms, which succeed to claim the static and dynamic slack time, with little overhead. Most of these papers deal with classical scheduling of task graph applications, which are not streaming applications. The techniques are similar, but the performance guarantee is a deadline on the total execution time. Rather, we consider workflow applications, i.e., several data sets must be processed by the task graph, and hence we bound the application period.

The problem of mapping workflow applications with the structure of a linear chain onto parallel platforms has already been widely studied, in particular on homogeneous platforms (see the pioneering paper [111]) and later for heterogeneous platforms [17]. These results are extended to account for energy consumption in [B7], where the target problem is mapping several linear chain applications on a fully interconnected platform, with three optimization criteria: power, period, and latency (execution time for one data set).

Mapping applications to chip multiprocessors. Many researchers have considered the mapping of tasks and threads to CMPs that are connected by a 2-dimentional network on a chip. The work in [7] introduces an approach to multi-objective exploration of mapping general task graphs to mesh-based CMPs using evolutionary algorithms. The approach is an efficient and accurate way to obtain the Pareto mappings that optimize performance and power consumption. In [64], an architecture-aware analytic mapping algorithm is presented. It uses a metric space that exactly captures the CMP topology to efficiently solve the problem. In [26], a compiler framework is presented to map the source code of an application to a mesh-based chip multiprocessor system. Compiler techniques are also used in [25] to dynamically change the speed of communication channels in CMPs to reduce energy consumption. In [2], the mapping of applications to heterogeneous multi-processor systems is performed by invoking runtime agents that are distributed among the processors. None of the above work considers the mapping of streaming applications onto CMP with the objective of minimizing power consumption while maintaining a specified throughput (period).

Summary. In this chapter, the application is a workflow whose structure is series-parallel task graph, and the goal is to map this application onto a CMP, with the objective of minimizing the energy consumption, given a threshold on the period of the workflow. We are therefore extending Chapter 2,

which was conducted for simpler application structures (linear chains instead of series-parallel graphs), and for a realistic platform (the CMP) instead of virtual cliques. To the best of our knowledge, this chapter investigates the complexity of this problem, and to propose practical solutions (polynomial time heuristics) for applications modeled by series-parallel graphs, which has never been done. The work in [124] shares the same objective as the work in this chapter but is purely empirical. It presents a two-phase heuristic for mapping a general acyclic graph onto a CMP by first assigning the levels of the graph to the rows of the CMP and then mapping the tasks in each level to the nodes of the row assigned to that level. The heuristic described in Section 4.5.3 follows a similar two-phase strategy but obeys the DAG-partition mapping rule (see Section 4.3.3).

4.3 Framework

In this section, we first describe the applicative framework (Section 4.3.1) and the target platform (Section 4.3.2). Then we detail the mapping strategies in Section 4.3.3. Finally, we formally define the bi-criteria optimization problem: we aim at minimizing the energy consumption while not exceeding a prescribed period. The formula to check that the period is not exceeded is given in Section 4.3.4, and the model for energy consumption is outlined in Section 4.3.5.

4.3.1 Applicative framework

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. In this study, the application is a series-parallel graph $\mathcal{G} = (\mathcal{S}, \mathcal{E})$, or SPG. Nodes of the graph correspond to different application stages, and are denoted by S_i, with $1 \leq i \leq n$, where $n = |\mathcal{S}|$ is the size of the graph. For each precedence constraint in the application, say from stage S_i to stage S_j, we have an edge $L_{i,j} \in \mathcal{E}$. For $1 \leq i \leq n$, w_i is the computation requirement of stage S_i, and for each $L_{i,j} \in \mathcal{E}$, with $1 \leq i, j \leq n$, $\delta_{i,j}$ is the volume of communication to be sent from S_i to S_j before S_j can start its computation.

A SPG is built from a sequence of compositions (parallel or series) of smaller-size SPGs. 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. For a parallel composition, the two sources are merged, as well as the two sinks (see Figure 4.1 for illustrative examples).

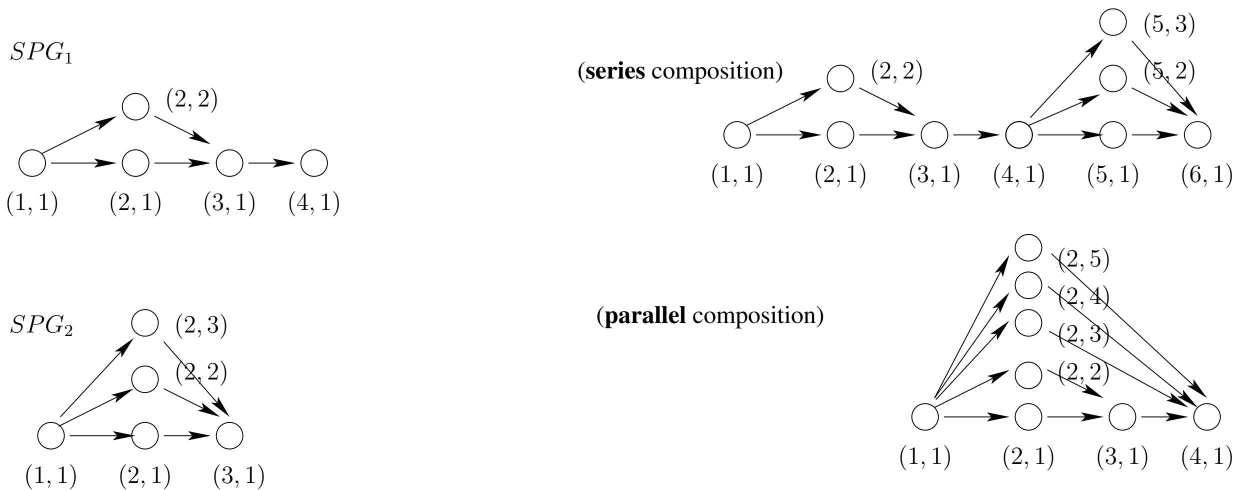


Figure 4.1: Examples of SPG composition.

We recursively define the *label* of each node in a SPG, which corresponds to its coordinates along a 2D-grid in the recursive construction: $\ell_i = (x_i, y_i)$ is the label of stage S_i, for $1 \leq i \leq n$. First, for a two-node SPG ($S_1 \to S_2$), the label of the source S_1 is $(1, 1)$, while the label of the sink S_2 is $(2, 1)$. The labels are then updated when composing the SPG. Consider two SPGs, SPG_1 with nodes $S_1^{(1)}, \dots, S_{n_1}^{(1)}$, and SPG_2 with nodes $S_1^{(2)}, \dots, S_{n_2}^{(2)}$, and their corresponding labels $\ell_i^{(1)} = (x_i^{(1)}, y_i^{(1)})$ and $\ell_j^{(2)} = (x_j^{(2)}, y_j^{(2)})$, for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$.

– For a serial composition, we merge the sink of SPG_1, $S_{n_1}^{(1)}$, with the source of SPG_2, $S_1^{(2)}$. The resulting SPG has $n = n_1 + n_2 - 1$ nodes with the following labels: for $1 \leq i \leq n_1$, $S_i = S_i^{(1)}$ and its label is $\ell_i = \ell_i^{(1)}$, and for $1 < j \leq n_2$, $S_{n_1+j-1} = S_j^{(2)}$ and the x values of the labels are incremented by $x_{n_1}^{(1)} - 1$, i.e., $\ell_{n_1+j-1} = (x_j^{(2)} + x_{n_1}^{(1)} - 1, y_j^{(2)})$.
– For a parallel composition, assume that $x_{n_1}^{(1)} \geq x_{n_2}^{(2)}$ (otherwise exchange the two SPGs, so that the first contains the longest path). We merge both sources ($S_1^{(1)}$ and $S_1^{(2)}$), and both sinks ($S_{n_1}^{(1)}$ and $S_{n_2}^{(2)}$). The resulting SPG has $n = n_1 + n_2 - 2$ nodes with the following labels: S_1 is the source and $\ell_1 = \ell_1^{(1)}$; S_n is the sink and $\ell_n = \ell_{n_1}^{(1)}$; for $1 < i < n_1$, $S_i = S_i^{(1)}$ and its label is $\ell_i = \ell_i^{(1)}$; for $1 < j < n_2$, $S_{n_1+j-2} = S_j^{(2)}$, and the y values of the labels are incremented by $y_{\max}^{(1)} = \max_{1 \leq i \leq n_1}(y_i^{(1)})$, i.e., $\ell_{n_1+j-2} = (x_j^{(2)}, y_j^{(2)} + y_{\max}^{(1)})$.

This construction is illustrated on the examples given in Figure 4.1. Note that these rules ensure that the source is always stage S_1, with $\ell_1 = (1, 1)$, and the sink is always stage S_n, with $\ell_n = (x_n, 1)$. Therefore, $\max_{1 \leq i \leq n} x_i = x_n$, and we denote by $y_{\max} = \max_{1 \leq i \leq n} y_i$ the maximum y value of the labels in the SPG, which we call *maximum elevation*. Intuitively, the maximum elevation denotes the maximal degree of parallelism of the SPG.

In the following, we focus the discussion on *bounded-elevation* SPGs, i.e., SPGs whose maximum elevation $y_{\max}$ is bounded by a constant. Indeed, dealing with bounded-elevation SPGs, rather than arbitrary SPGs, or even arbitrary DAGs, is a trade-off between tractability and generality. On the one hand, bounded-elevation SPGs correspond to a wide spectrum of applications, and nicely generalize linear chains and trees (a tree can easily be transformed into a SPG by adding fake nodes mirroring the tree). For instance, all the benchmarks of the *StreamIt* suite [109] are bounded-elevation SPGs: their maximum elevations range from $y_{\max} = 1$ (linear chain) to $y_{\max} = 17$. On the other hand, the problem of mapping a simple fork-join graph with n nodes (unbounded-elevation graph) onto two processors, in order to minimize the energy given a period bound, is NP-complete (reduction from 2-PARTITION, see Section 4.4.1). Dealing with bounded-elevation graphs enables us to identify polynomial instances, thus providing optimal solutions for some problem instances.

4.3.2 Platform

The target platform is a CMP (**C**hip **M**ulti**P**rocessor), composed of $p \times q$ homogeneous cores $\mathcal{C}_{u,v}$, with $1 \leq u \leq p$, $1 \leq v \leq q$, arranged along a rectangular grid. There is a vertical (internal and bi-directional) communication link between $\mathcal{C}_{u,v}$ and $\mathcal{C}_{u+1,v}$, for $1 \leq u \leq p-1$, $1 \leq v \leq q$, and a horizontal link between $\mathcal{C}_{u,v}$ and $\mathcal{C}_{u,v+1}$, for $1 \leq u \leq p$, $1 \leq v \leq q-1$. All links have the same bandwidth BW (in each direction). This means that it takes a time $\frac{\delta}{BW}$ to send δ bytes from one processor to a neighboring processor. It is possible to use only some of the communication links, and for instance to configure the $p \times q$ CMP as a $1 \times pq$ bi-directional linear array, called *bi-directional uni-line CMP*.

Although the cores of a CMP share the same memory space, it is possible to implement the message passing models on CMPs [83] by writing and reading from shared memory locations. However, for

scalability purpose, CMPs with large number of cores will be organized as a mesh of tiles, each with its own cache [68]. Therefore, communication through shared memory ultimately translates to exchange of coherence traffic between the tiles [54, 34, 59]. Specifically, in the streaming model assumed in this chapter, when a stage S_i, mapped to a core $\mathcal{C}_{u,v}$, writes into a shared variable, X, that shared variable is cached in the local cache of $\mathcal{C}_{u,v}$. Then, when a stage S_j with $L_{i,j} \in \mathcal{E}$, mapped to a core $\mathcal{C}_{u',v'} \neq \mathcal{C}_{u,v}$, reads X, the cache coherence protocol guarantees that X is cached in the local cache of $\mathcal{C}_{u',v'}$. Therefore, the values of the cache line containing X is sent from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v'}$. In other words, if two stages S_i and S_j, connected with an edge $L_{i,j}$, are mapped onto two distinct processors, a communication of size $\delta_{i,j}$ must occur (implicit messages) to keep the cached values coherent [1]. Hence, irrespectively of the programming model used to implement the SPG, the weight on a directed edge between two nodes in the SPG represents the volume of communication to be sent between the cores executing the corresponding application stages.

As mentioned in Section 4.2, the voltage and frequency of each core of the CMP can be set to different values. Altogether, there is a set of possible supply voltages, together with a set of possible frequencies (or modes, or speeds), for each core. Let $\mathsf{S} = \{s^{(1)}, \dots, s^{(m)}\}$ denote the set of all possible speeds. It takes a time $\frac{w_i}{s^{(k)}}$ to execute one data set for stage S_i at speed $s^{(k)} \in \mathsf{S}$ on a given core. Each speed induces a different dynamic power consumption, as discussed in Section 4.3.5 below.

4.3.3 Mapping strategies

We discuss several mapping rules to map the SPG application onto the CMP. As for the application graph, we use *DAG-partition* mappings, which represent a trade-off between *one-to-one* and *general* mappings. The rationale is the following. One-to-one mappings obey the simplest rule: each application stage is mapped onto a distinct core. While easier to optimize and implement, this rule may be unduly restrictive, and is likely to lead to high communication costs. Obviously, it also requires that $p \times q \geq n$, thereby limiting its applicability to large platforms or small applications. A natural extension is to search for DAG-partition mappings: we first partition the initial SPG into subsets, or clusters, such that the resulting graph is acyclic. Hence this mapping rule states that if two stages S_i and S_j are in the same subset of the partition, then any other stage S_k which has an incoming dependency from S_i and an outgoing dependency to S_j, must be in the same subset of the partition. Then we map the subsets of the partition onto the cores in a one-to-one fashion. Using this mapping rule, a core which is executing a subset I of stages $\{S_i\}_{i \in I}$ will perform at most one input and one output communication for each elevation value $\{y_i\}_{i \in I}$. This is well in accordance with our initial assumption that the SPG has bounded elevation $y_{\max}$, because it ensures that each core has at most $y_{\max}$ communications to perform at each period. In contrast, a fully general mapping, that allow for arbitrary partitions of the original application graph, would require an arbitrary number of communications, only bounded by the total number of stages n, hence an unlimited amount of buffer space. Moreover, even for bounded-elevation SPGs, the problem of finding the general mapping which minimizes the energy given a period bound is trivially NP-complete (linear chain onto two processors, reduction from 2-PARTITION [44]).

Formally, the mapping is defined by an allocation function

$$alloc : \{1, \dots, n\} \to \{1, \dots, p\} \times \{1, \dots, q\} ,$$

which maps stages onto cores. In other words, if stage S_i is mapped onto core $\mathcal{C}_{u,v}$, we have $alloc(i) = (u, v)$. Once application stages are mapped onto cores, there remains to decide how to route communications between two cores which need to communicate because of the stage assignment. Therefore, for

1. It is assumed that the cache coherence protocol supports cache-to-cache transfer and exploits communication locality by tracking in each core the location of frequently accessed blocks [53].

each application edge $L_{i,j} \in \mathcal{E}$, if $alloc(i) \neq alloc(j)$, we define $path_{i,j}$ as the set of communication links that are used to communicate from core $alloc(i)$ to core $alloc(j)$. Note that these paths must be defined for the mapping to be fully determined.

4.3.4 Period

As motivated above, we assume that data sets arrive at regular time intervals, which is called the *period* of the application, and denoted by T. Then, given a mapping and an execution speed for each core, we can check whether the application can be executed at the prescribed rate: we must ensure that the cycle-time of each resource (computation or communication link) does not exceed T.

Let $w_{u,v} = \sum_{1 \leq i \leq n | alloc(i)=(u,v)} w_i$ be the total amount of work assigned to core $\mathcal{C}_{u,v}$, running at speed $s_{u,v} \in \mathsf{S}$. The cycle-time of $\mathcal{C}_{u,v}$ for computations is $w_{u,v}/s_{u,v}$. For communications, $b_{(u,v)\to(u',v')} = \sum_{1 \leq i,j \leq n | (u,v)\to(u',v') \in path_{i,j}} \delta_{i,j}$ is the number of bits sent from $\mathcal{C}_{u,v}$ to a neighbor core $\mathcal{C}_{u',v'}$ [2]. The cycle-time of the communication link $(u,v) \to (u',v')$ is $b_{(u,v)\to(u',v')}/BW$.

We can then compute the maximum cycle-time, which is the maximum cycle-time of all resources, and check that it is not greater than T.

4.3.5 Energy model

Once a SPG application has been mapped onto the CMP, there are two sources of energy consumption: the cores consume energy for computations and the routers consume additional energy for communications.

For the computations, we refine the model of Chapter 2: each core involved in the execution consumes some static energy during the whole period T, and some dynamic energy that depends on the amount of operations, and on the speed at which these operations are executed. Let $\mathcal{A}$ be the set of active cores: $\mathcal{A} = \{\mathcal{C}_{u,v},\ 1 \leq u \leq p, 1 \leq v \leq q \mid \exists\, 1 \leq i \leq n,\ alloc(i) = (u,v)\}$. For each core $\mathcal{C}_{u,v} \in \mathcal{A}$, let $w_{u,v}$ be its assigned work and $s_{u,v}$ its speed. The total energy consumed for computations is

$$E^{(\text{comp})} = |\mathcal{A}| \times P^{(\text{comp})}_{\text{leak}} \times T + \sum_{\mathcal{C}_{u,v} \in \mathcal{A}} \frac{w_{u,v}}{s_{u,v}} \times P^{(\text{comp})}_{s_{u,v}} \,,$$

where T is the prescribed period, $P^{(\text{comp})}_{\text{leak}}$ is the leakage power dissipated together with computations, and $P^{(\text{comp})}_{s_{u,v}}$ is the dynamic power associated with speed $s_{u,v}$. Particularly, we can use the formula $P^{(\text{comp})}_{s_{u,v}} = s^{\alpha}_{u,v}$ of the previous chapters, for any α value.

For the communications, there is also a static part due to leakage, which is paid for all cores: even if a core is not enrolled in the computation, its routers and communication links may be used to route data between remote processors. The dynamic part is directly proportional to the number of bits that are sent across each link. Hence,

$$E^{(\text{comm})} = P^{(\text{comm})}_{\text{leak}} \times T + \left(\sum_{u,v} \sum_{u',v'} b_{(u,v)\to(u',v')} \right) \times E^{(\text{bit})} \,,$$

where T is the period, $P^{(\text{comm})}_{\text{leak}}$ is the aggregated leakage power dissipated by all routers and links, and $E^{(\text{bit})}$ is the energy to transfer a bit across neighboring cores. Finally, the total energy consumption is $E = E^{(\text{comp})} + E^{(\text{comm})}$.

2. (u'=$u+1$ and v'=v) or (u'=$u-1$ and v'=v) or (u'=u and v'=$v+1$) or (u'=u and v'=$v-1$).

We are ready to formally define the optimization problem:

Definition 4.1 (MINENERGY(T)). *Given a bounded-elevation SPG and a period threshold T, find a mapping whose maximal cycle-time does not exceed T and whose energy E is minimum.*

4.4 Complexity results

In this section, we assess the complexity of the MINENERGY(T) problem for various instances. We classify results depending upon the target CMP, which may be uni-directional uni-line (see Section 4.4.1), bi-directional uni-line (see Section 4.4.2), or bi-directional 2D mesh (see Section 4.4.3). The only polynomial instance of MINENERGY(T) is for the uni-directional uni-line CMP. In this case, we exhibit a dynamic programming algorithm that finds the optimal solution. It is worth noting that this polynomial instance becomes NP-complete for SPGs of unbounded elevation. All other problem instances are NP-hard, and we formulate the problem as an integer linear program in Section 4.4.4.

4.4.1 Uni-directional uni-line CMP

In this section, we assume that the CMP is configured as a uni-directional linear array of q processors. First we provide a polynomial algorithm to solve the case of bounded-elevation SPGs. As a digression from the main focus of this chapter (bounded-elevation SPGs), we prove that the problem becomes NP-hard for SPGs of unbounded elevation.

Theorem 4.1. *The* MINENERGY*(T) problem on a uni-directional uni-line CMP has polynomial complexity.*

Proof. We exhibit a dynamic programming algorithm which computes the optimal solution. Let $\mathcal{G}$ be a bounded-elevation SPG. First we define *admissible* subgraphs of $\mathcal{G}$ recursively:

- $\mathcal{G}$ is admissible;
- if a subgraph G of $\mathcal{G}$ is admissible, then any subgraph of G obtained by deleting one node which has no successor in G is admissible too.

Let H be a set of one or several nodes deleted from G with this process, and let $G' = G \setminus H$. Note that the partition $\{G', H\}$ is acyclic, and that any possible acyclic partition of G into two subgraphs can be obtained with this construction. If we iterate the construction on G', we can build any DAG-partition of $\mathcal{G}$.

How many admissible subgraphs can we have? Let $y_{\max}$ be the maximal elevation of $\mathcal{G}$. Consider any admissible subgraph G. By definition, two nodes with the same y coordinate are linked by a dependence. Therefore, for each value of y between 1 and $y_{\max}$, there can be at most one node of elevation y and without successor in G. Hence there are at most $n^{y_{\max}}$ admissible subgraphs (and this bound is asymptotically met for a fork-join shaped graph composed of $y_{\max}$ chains of length $n/y_{\max}$ assembled with a source and sink node).

For any admissible subgraph G of $\mathcal{G}$, let $\mathcal{E}(G, k)$ be the minimum energy consumption required to execute the subgraph G onto exactly the first k processors. The goal is to determine $\min_{k=1}^{q} \mathcal{E}(\mathcal{G}, k)$.

The dynamic programming formulation can be expressed as:

$$\mathcal{E}(G, k) = \min_{G' \subseteq G} \left(\mathcal{E}(G', k-1) \oplus \mathcal{E}^{\text{cal}}(G \setminus G') \right) \ ,$$

with the initialization $\mathcal{E}(G, 1) = \mathcal{E}^{\text{cal}}(G)$.

The minimum is taken over all admissible subgraphs G' such that communications between G' and $G \setminus G'$ do not exceed the bandwidth: $\frac{C^{\text{out}}(G')}{BW} \leq T$, where $C^{\text{out}}(G')$ denotes the aggregated output data volume of G', i.e., the sum of the output data δ_i of all stages $S_i \in G'$ which have no successor in G'.

$\mathcal{E}^{\text{cal}}(H)$ represents the energy consumed for the computations of the nodes in H when mapped to the same processor. Given such a node set H, we select the minimum speed that allows for computing all the stages in H within the period T, and we compute the corresponding energy consumption. If no such speed exists, we let $\mathcal{E}^{\text{cal}}(H) = +\infty$. Finally, the $\oplus$ operator means that the energy consumed by the induced communications is added to the sum.

At each step, there are no more than $n^{y_{\max}}$ admissible graphs G', and therefore we have at most $n^{2y_{\max}}$ values of $\mathcal{E}^{\text{cal}}(H)$ to compute, which is done in $O(n)$. Altogether, we have designed an algorithm whose worst-case complexity is $O(q \times n \times n^{2y_{\max}})$, which is polynomial since $y_{\max}$ is a constant. ■

The previous theorem only holds for bounded-elevation SPGs. With unbounded-elevation SPGs, the problem becomes NP-hard:

Proposition 4.1. *The extension of* MINENERGY(T) *to unbounded-elevation SPGs on a uni-directional uni-line CMP is NP-complete.*

Proof. In fact, without any energy consideration, the simpler mono-criterion problem of matching a prescribed period is NP-complete. The associated decision problem is as follows: given a period T, is there a DAG-partition mapping whose period is no more than T? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time that it is valid by computing its period.

To establish the completeness, we use a reduction from 2-PARTITION [44]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given n strictly positive integers $a_1, a_2, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$.

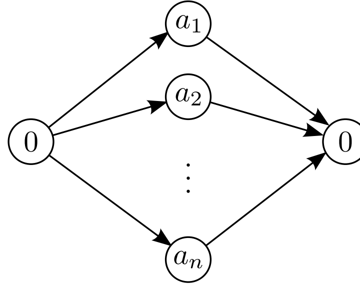


Figure 4.2: Unbounded-elevation SPG for the uni-directional uni-line CMP proof.

We build an instance $\mathcal{I}_2$ of our problem: the application consists of a fork-join graph of elevation n, as illustrated in Figure 4.2. We denote by S_0 the source node, S_{n+1} the sink node, and S_i, for $1 \leq i \leq n$, is the ith node of the fork-join. For computation costs, we have $w_0 = w_{n+1} = 0$, and $w_i = a_i$, and there are no communication costs. The platform consists of two cores which can operate only at a unique speed $s = 1$. Finally, we ask whether we can achieve a period $\frac{S}{2}$.

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. The equivalence between both problems is straightforward: if $\mathcal{I}_1$ has a solution I, then we assign S_0 and $\{S_i\}_{i \in I}$ to the first core, S_{n+1} and $\{S_i\}_{i \notin I}$ to the second core. The mapping is a DAG-partition, and its period is $\frac{S}{2}$, therefore we find a solution to $\mathcal{I}_2$. On the other hand, if $\mathcal{I}_2$ has a solution, the period on each core must be exactly $\frac{S}{2}$ because the total computation requirement is S, and therefore we have a 2-partition of the stages S_i, for $1 \leq i \leq n$. This concludes the proof. ■

4.4.2 Bi-directional uni-line CMP

Theorem 4.2. *The* MINENERGY*(T) problem on a bi-directional uni-line CMP is NP-complete.*

Proof. As for Proposition 4.1, the simpler mono-criterion problem of matching a prescribed period T, without any energy consideration, is NP-complete. However, the reduction proof becomes quite involved, since we consider a bounded-elevation SPG.

The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time that it is valid by computing its period. To establish the completeness, we use a reduction from 2-PARTITION [44]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given n strictly positive integers $a_1, a_2, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$.

We build an instance $\mathcal{I}_2$ of our problem: the bounded-elevation SPG of the application is represented in Figure 4.3. There are $3n + 3$ stages, computation costs of each stages are equal to 1, and communication costs are depicted in the figure. The platform is a bi-directional uni-line CMP of $1 \times q$ cores, where $q = 3n + 3$. Each core can operate only at a unique speed $s = 1$, and the bandwidth of each link is $BW = 3S/2 + \epsilon$. Finally, we ask whether we can achieve a period of 1. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does. First note that any solution of $\mathcal{I}_2$ is a one-to-one mapping, because of the constraint on the period and the computation costs of stages. Indeed, if two or more stages were mapped onto the same core, the period would be at least 2.

Assume first that $\mathcal{I}_1$ has a solution, I. We assume that $I = \{i_1, \ldots, i_k\}$ and $\bar{I} = \{1, \ldots, n\} \setminus I = \{\bar{i}_1, \ldots, \bar{i}_{n-k}\}$, with $\sum_{j=1}^{k} a_{i_j} = \sum_{j=1}^{n-k} a_{\bar{i}_j} = S/2$. For $\mathcal{I}_2$, we map the application graph onto the CMP as illustrated in Figure 4.3: for all $j \in \{1, \ldots, k\}$, C_j is mapped onto $\mathcal{C}_{2j-1}$ and B_j onto $\mathcal{C}_{2j}$. Then the stage In is mapped onto $\mathcal{C}_{2k+1}$, for all $j \in \{1, \ldots, n+1\}$, A_j is mapped onto $\mathcal{C}_{2k+1+j}$, and the stage Out is mapped onto $\mathcal{C}_{2k+n+3}$. Finally for all $j \in \{1, \ldots, n-k\}$, B_j is mapped onto $\mathcal{C}_{2k+n+2+2j}$, and C_j onto $\mathcal{C}_{2k+n+2j+3}$. The mapping is one-to-one so that the period constraint is fulfilled for computations. We now show that, on each link, the sum of communications does not exceed BW, and hence the bound on the period is not violated.

Let us first consider the link $\ell^{(h)}_{2k+1+j}$, with $j \in \{1, \ldots, n\}$: the amount of communications on this link is equal to $S/2 + \epsilon$ (communication from A_j to A_{j+1}), plus at most $\sum_{i=1}^{n} a_i = S$ (communications from A_i to B_i), therefore a total of no more than $3S/2 + \epsilon = BW$.

Then we consider the link $\ell^{(h)}_{2j-1}$, with $j \in \{1, \ldots, k\}$: the amount of communications is then $S + \epsilon$ (from B_{i_j} to C_{i_j}) plus at most $\sum_{i \in I} a_i = S/2$ (communications from A_i to B_i, for $i \in I$), which is no more than BW. Finally, on the link $\ell^{(h)}_{2j}$, with $j \in \{1, \ldots, k\}$, there are at most $\sum_{i \in I} a_i = S/2$

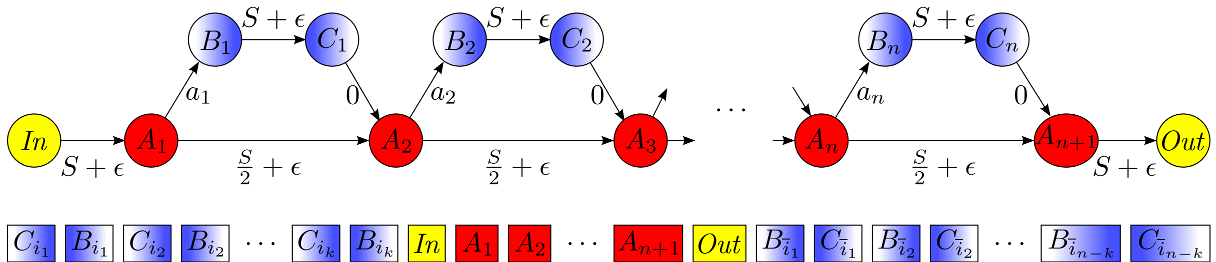


Figure 4.3: Bounded-elevation SPG and mapping for the bi-directional uni-line CMP proof.

communications (from A_i to B_i, for $i \in I$). Similarly, we can prove that the bandwidth is not exceeded on link $\ell^{(h)}_{2k+n+2+j}$, for $j \in \{1, \dots, 2(n-k)\}$.

Only two links remain now: $\ell^{(h)}_{2k+1}$ and $\ell^{(h)}_{2k+n+2}$. The only communications not equals to 0 that go through $\ell^{(h)}_{2k+1}$ are the communications from A_i to B_i, for $i \in I$, thus the link bandwidth constraint is fulfilled. This holds true for $\ell^{(h)}_{2k+n+2}$, reasoning with $\bar{I}$ instead of I. We conclude that $\mathcal{I}_2$ has a solution.

Assume now that $\mathcal{I}_2$ has a solution. We prove that the mapping is necessary similar to that of Figure 4.3, and that stages B_i and C_i must be 2-partitioned.

Let σ be the permutation of $\{1, \dots, n+1\}$ such that, for each $i \in \{1, \dots, n\}$, the core assigned to $A_{\sigma(i)}$ is to the left of the one assigned to $A_{\sigma(i+1)}$. First, let us assume that there exists $i^{(0)} \in \{1, \dots, n\}$ such that the stage In is mapped between $A_{\sigma(i^{(0)})}$ and $A_{\sigma(i^{(0)}+1)}$. Since there is a path (with edges of weight $S/2 + \epsilon$) going through all the $A_{\sigma(i)}$, a communication of size $S/2 + \epsilon$ occurs on all links between the core assigned to $A_{\sigma(i^{(0)})}$ and the core assigned to $A_{\sigma(i^{(0)}+1)}$. Because of the mapping of In, we deduce that there is a link on which the amount of communications is at least $3S/2 + 2\epsilon$, which leads to a contradiction. Therefore, we showed that the core that is assigned to In is either to the left of the core assigned to $A_{\sigma(1)}$ or to the right of the core assigned to $A_{\sigma(n+1)}$. The same result holds for Out (similar proof).

Moreover note that In and Out cannot be on the same side, otherwise either the link after the core assigned to $A_{\sigma(n+1)}$ or the link before the core assigned to $A_{\sigma(1)}$ would transmit at least two communications of size $S + \epsilon$. We can assume, without loss of generality that In is mapped on the left, and Out on the right.

Similarly, for all $i \in \{1, \dots, n\}$, B_i and C_i cannot be mapped onto a core between the core assigned to a $A_{\sigma(i')}$ and the one assigned to $A_{\sigma(i'+1)}$, or In and $A_{\sigma(1)}$, or $A_{\sigma(n+1)}$ and Out. The B_i are thus mapped either to the left of In, or to the right of Out, similarly to Figure 4.3.

Finally, let I be a subset of $\{1, \dots, n\}$ such that $i \in I$ if and only if B_i is mapped to the left of In. Then, since the bandwith bound is not exceeded between the core assigned to In and the one assigned to $A_{\sigma(1)}$ on one hand, and between $A_{\sigma(1)}$ and Out on the other hand, we have necessarily $\sum_{i \in I} a_i + S + \epsilon \leq 3S/2 + \epsilon$ and $\sum_{i \notin I} a_i + S + \epsilon \leq 3S/2 + \epsilon$. Therefore, $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S/2$, $\mathcal{I}_1$ has a solution, which concludes the proof. ■

4.4.3 Square CMP

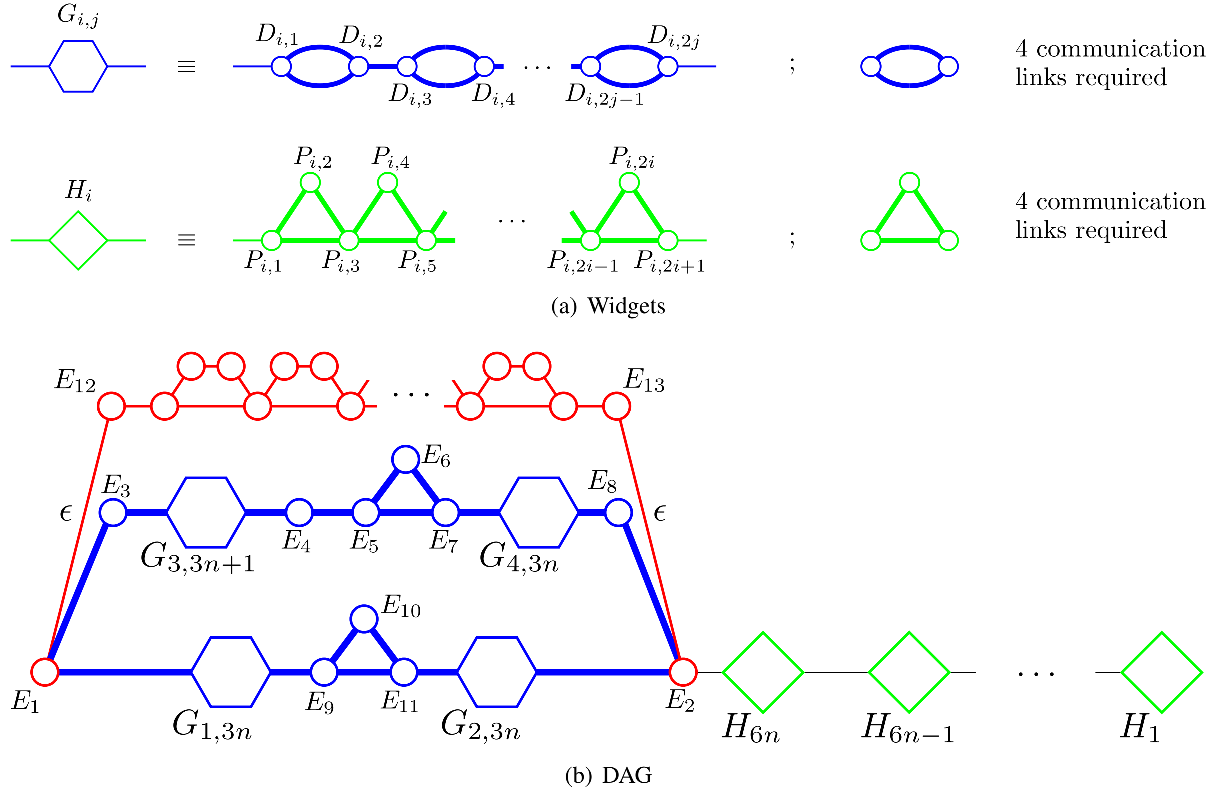
In this section, we focus on square CMPs. We know from Theorem 4.2 that the problem is NP-hard for a $1 \times q$ CMP, hence for CMPs of arbitrary shapes. However, the problem complexity for a square CMP of size $p \times p$ is not a consequence of Theorem 4.2. We now establish this complexity:

Theorem 4.3. *The* MINENERGY(T) *problem on a square CMP is NP-complete.*

Proof. As for Theorem 4.2, the simpler mono-criterion problem of matching a prescribed period T, without any energy consideration, is NP-complete. The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time that it is valid by computing its period.

To establish the completeness, we use a reduction from 2-PARTITION [44]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given $3n+1$ strictly positive integers $a_1, a_2, \dots, a_{3n+1}$, does there exist a subset I of $\{1, \dots, 3n+1\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{3n+1} a_i$.

We build the following instance $\mathcal{I}_2$ for our problem, re-using the construction proposed in Theorem 4.2. The CMP is composed of $p \times p$ cores with a single speed 1, linked with a bandwidth $BW = 3S/2 + \epsilon$, where $p = 6n + 4$. The series-parallel graph is described as a directed acyclic



(a) Widgets

(b) DAG

Figure 4.4: Problem instance

graph (DAG) in Figure 4.4(b), using some widgets introduced in Figure 4.4(a). To transform this DAG into a SPG, we use the transformation explained in Figure 4.5: the blue nodes in widgets G can be replaced by two nodes with computation cost $1/2$, which must be mapped onto the same core because of bandwidth constraints. All computation costs in the DAG are equal to 1, and in the following we conduct the reasoning on the DAG. The size of blue and green communications is equal to the bandwidth BW, and there is no communication between two H_i widgets, neither between E_2 and H_{6n}. The size of communications from E_1 to E_{12} on the one side, and from E_{13} to E_2, on the other side, is equal to ϵ. The subgraph between E_{12} and E_{13} is the graph of Figure 4.3, replacing n by $3n+1$. Finally, we ask whether we can achieve a period of 1. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$.

We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does. First note that any solution of $\mathcal{I}_2$ is a one-to-one mapping, because of the constraint on the period and the computation costs of stages. Indeed, if two or more stages were mapped onto the same core, the period would be at least 2.

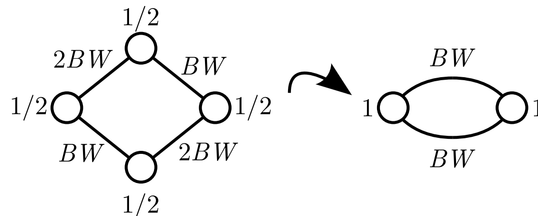


Figure 4.5: SPG to DAG.

Assume first that $\mathcal{I}_2$ has a solution. We show that the red nodes are necessary mapped onto a linear chain of cores, and communications never escape out of this linear chain.

In each widget $G_{i,j}$, the two communications between $D_{i,2k-1}$ and $D_{i,2k}$, for $k \in \{1,\ldots,j\}$, must occur on at least 4 links and no other communications not equal to 0 can use those links, because one communication fills entirely a link, and the mapping is one-to-one. In the same way, in every widget H_i, the three communications between $P_{i,2k-1}$, $P_{i,2k}$ and $P_{i,2k+1}$, for $k \in \{1,\ldots,i\}$, must occur on at least 4 links and no other communication (not equal to 0) can use those links. Moreover, there are 19 more communications of size BW, which thus require at least 19 more communication links. Altogether, we need at least:

$$\sum_{i=1}^{6n} 4i + 3 \times (15n-1) + (15n+4) + 19 = 2(6n)(6n+1) + 60n + 20$$
$$= 72n^2 + 72n + 20$$
$$= 2p(p-1) - 2(p-2)$$

communication links to map all communications except the red ones. If we use more links to map blue or green communications, there would be at most $2(p-2)-1$ free remaining links. Now the graph contains $2(p-2)+1$ red nodes, thus a red node would be isolated (i.e., it would have no available communication link), which is not possible, because each red node must communicate with at least one other red node. Thus, we have exactly $2(p-2)$ communication links for the red communications and $2p \times (p-1) - 2(p-2)$ communication links for blue and green communications.

The blue nodes of degree 3 are on the border of the CMP: those nodes cannot be mapped onto a corner core, because they need at least 3 free communication links, and they cannot be mapped onto a middle core either. In this case, three of four communication links would be indeed used, and the remaining empty communication link could not be used by another communication: an incoming communication could not exit through another link. The nodes $P_{3,1},\ldots,P_{3,p-2}$ of the widget $G_{3,3n+1}$ must be mapped in order on a border, otherwise we lose at least one communication link. Without loss of generality, we can assume that they are mapped respectively onto cores $\mathcal{C}_{2,1},\ldots,\mathcal{C}_{p-1,1}$. The communication between $P_{3,1}$ and $P_{3,2}$ must occur on links $\ell^{(h)}_{2,1}$, $\ell^{(v)}_{2,2}$ and $\ell^{(h)}_{3,1}$ in order not to lose any communication link. In the same way, the communication between $P_{3,p-3}$ and $P_{3,p-2}$ takes the links $\ell^{(h)}_{p-2,1}$, $\ell^{(v)}_{p-2,2}$ and $\ell^{(h)}_{p-1,1}$. As a result, again from the fact that we cannot lose any communication link, E_4 is mapped onto $\mathcal{C}_{p,1}$,

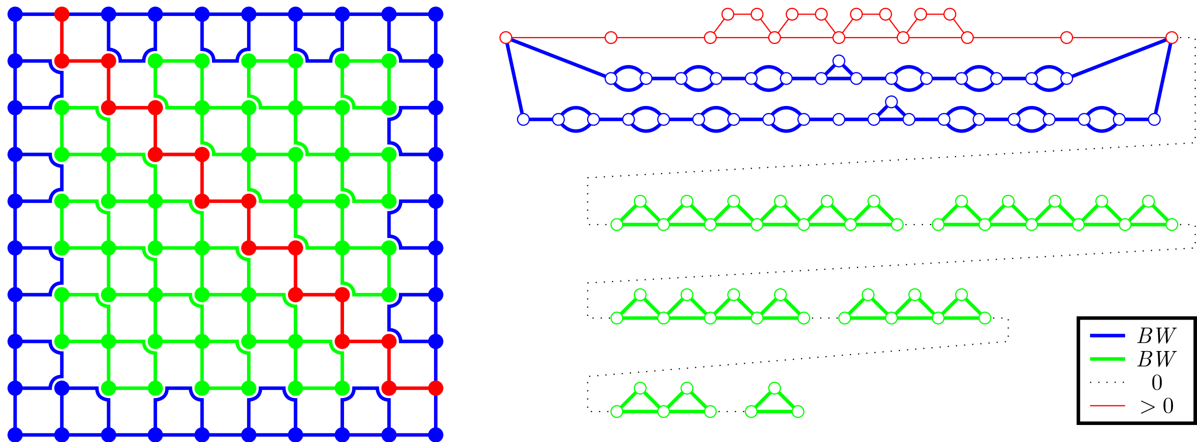


Figure 4.6: Example 10×10.

E_3 onto $\mathcal{C}_{1,1}$ then E_1 onto $\mathcal{C}_{1,2}$. In the same manner again, nodes $E_5, E_7, P_{4,1}, \ldots, P_{4,p-2}$ are mapped respectively onto cores $\mathcal{C}_{p,2}, \ldots, \mathcal{C}_{p,p-1}$, E_8 onto $\mathcal{C}_{p,p}$ and E_2 onto $\mathcal{C}_{p-1,p}$.

If the graph composed of the cores on which a red node is mapped, linked with the communication links where a red communication occurs, is not a chain, E_1 and E_2 cannot be connected, because there are only $2(p-2)$ remaining communication links and the Manhattan distance between E_1 and E_2 is $2(p-2)$. Since the 2 additional nodes E_1 and E_2, and the two communications of weight ϵ do not change anything, we are in the case of proof of Theorem 4.2. Therefore $\mathcal{I}_1$ has a solution.

We assume now that $\mathcal{I}_1$ has a solution. We give the mapping for $n = 1$ in Figure 4.6, which can convince us that such a mapping, where red nodes and communications are mapped onto a linear chain, can be found for any $n > 1$. Then we use again the proof of Theorem 4.2 to conclude that $\mathcal{I}_2$ has a solution, and hence conclude the proof. ■

4.4.4 Integer linear program

The general problem of finding the optimal DAG-partition mapping, for a given period, has been shown to be NP-hard. However, we formulate in this section the problem as an integer linear program (ILP), which allows us to find the optimal solution of the problem (in exponential time) for small problem instances. Actually, this ILP can also find the optimal general mapping (without the restriction of DAG-partition mappings), by removing the DAG-partition constraint from the program.

Unfortunately, because of the large number of variables needed to express communication paths in the CMP, we were unable to obtain results on a platform larger than a 2×2 CMP with ILOG CPLEX [38].

Constants

We first define the set of constant values that define our problem. The application is composed of n stages $S_1, \ldots, S_n$, and a set of edges $\mathcal{E}$:

- for $1 \le i \le n$, $w(i)$ is the amount of computations of node S_i, i.e., it corresponds to the w_i parameter;
- for $1 \le i, j \le n$, $\ell(i,j) = 1$ if there is a link between S_i and S_j (i.e., if $L_{i,j} \in \mathcal{E}$), and then $\delta(i,j)$ is the amount of communications between the two stages (it corresponds to the $\delta_{i,j}$ parameter); otherwise $\ell(i,j) = \delta(i,j) = 0$;
- we define ℓ^* as the transitive closure of ℓ, i.e., for $1 \le i,j \le n$, $\ell^*(i,j) = 1$ if there is a dependence path from S_i to S_j, otherwise $\ell^*(i,j) = 0$.

For the platform, we consider a $p\times q$ CMP, and we need to compute beforehand the energy consumed by a core when running at any speed.

- for $1 \le k \le m$, $s(k)$ is the k-th possible speed of a core;
- for $1 \le k \le m$, $Estat = P_{\text{leak}}^{(\text{comp})} \times T$ (static energy consumption for one core);
- for $1 \le k \le m$, $Edyn(k) = P_{s(k)}^{(\text{comp})}/s(k)$ (it must be multiplied by the amount of computation on the core to return the dynamic energy consumption, see Section 4.3.5).

Finally, BW is the link bandwidth, and T is the bound on the period.

Variables

Now that we have defined the constants that define our problem, we define unknown variables to be computed:

- for $1 \le i \le n$, $1 \le k \le m$, $1 \le u \le p$ and $1 \le v \le q$, $x_{i,k,u,v}$ is a boolean variable equal to 1 if stage S_i is mapped onto core $\mathcal{C}_{u,v}$, operated at speed $s(k)$, and 0 otherwise; there are $n \times m \times p \times q$ such variables;

– for $1 \leq k \leq m$, $m_{k,u,v}$ is a boolean variable equal to 1 if core $\mathcal{C}_{u,v}$ is operated at speed $s(k)$, and 0 otherwise; there are $m \times p \times q$ such variables;
– for $1 \leq i, j \leq n$, $1 \leq u \leq p$ and $1 \leq v \leq q$, $c^N_{i,j,u,v}$ (resp. $c^S_{i,j,u,v}$, $c^W_{i,j,u,v}$ and $c^E_{i,j,u,v}$) is a boolean variable equal to 1 if there is a communication for link $L_{i,j}$ between core $\mathcal{C}_{u,v}$ and its *north* (resp. *south*, *west*, *east*) neighbor $\mathcal{C}_{u-1,v}$ (resp. $\mathcal{C}_{u+1,v}$, $\mathcal{C}_{u,v-1}$, $\mathcal{C}_{u,v+1}$) and 0 otherwise; for $u = 1$ (resp. $u = p$, $v = 1$, $v = q$), we enforce that the variable is set to 0 (no possible communication because of the borders of the CMP); there are $4 \times n^2 \times p \times q$ such variables.

For convenience, we note $c^+_{i,j,u,v} = c^N_{i,j,u,v} + c^S_{i,j,u,v} + c^W_{i,j,u,v} + c^E_{i,j,u,v}$.

Constraints

Finally, we must write all constraints involving our constants and variables. In the following, unless stated otherwise, i, j, i' span $\{1, \ldots, n\}$ (stage indices); u, u' span $\{1, \ldots, p\}$ and v, v' span $\{1, \ldots, q\}$ (processor indices), and finally k, k' span $\{1, \ldots, m\}$ (speed, or mode indices). First we need constraints to guarantee that the allocation of stages to cores is a valid allocation, and that the speed of each core is correctly set.

– $\forall i, k,\ \sum_{u,v} x_{i,k,u,v} = 1$: each stage is allocated to exactly one core;
– $\forall k, u, v,\ m_{k,u,v} \geq \sum_i x_{i,k,u,v}$: if stage S_i is mapped onto $\mathcal{C}_{u,v}$ operated at speed $s(k)$, then $\mathcal{C}_{u,v}$ must be operated at speed $s(k)$;
– $\forall u, v\ \sum_k m_{k,u,v} \leq 1$: each core is operated at no more than one speed (either the core is on and the sum equals 1, or it is off and the sum equals 0).

Then, we need to ensure that communications are correctly scheduled, by enforcing constraints on the $c_{i,j,u,v}$ variables.

– $\forall i, j, u, v,\ c^N_{i,j,1,v} = 0$, $c^S_{i,j,p,v} = 0$, $c^W_{i,j,u,1} = 0$, and $c^E_{i,j,u,q} = 0$: no communication is allowed outside the borders of the CMP;
– $\forall i, j, u, v,\ c^+_{i,j,u,v} \leq \ell(i,j)$: there is no communication from S_i to S_j if there is no dependence constraint between these two stages;
– $\forall i, j, k, u, v,\ x_{i,k,u,v} + x_{j,k,u,v} + c^+_{i,j,u,v} \leq 2$: this condition enforces that if S_i and S_j are mapped onto the same core, $\mathcal{C}_{u,v}$, then there is no communication for link $L_{i,j}$ initiated from $\mathcal{C}_{u,v}$;
– $\forall i, j, k,\ c^+_{i,j,u,v} \geq x_{i,k,u,v} + \sum_{k',(u,v)\neq(u',v')} x_{j,k',u',v'} + \ell(i,j) - 2$: this initiates the communication for $L_{i,j}$ if S_i and S_j are mapped onto two distinct cores; the communication must occur into one of the directions (N,S,W or E);
– $\forall i, j, u < p, v,\ c^S_{i,j,u,v} \leq c^+_{i,j,u+1,v} + \sum_k x_{j,k,u+1,v} \leq 2 - c^S_{i,j,u,v}$: if there was a communication initiated from $\mathcal{C}_{u,v}$ to the south for $L_{i,j}$ ($c^S_{i,j,u,v} = 1$), then either we reach the destination core ($\sum_k x_{j,k,u+1,v} = 1$), or the communication must be forwarded on one of the links from $\mathcal{C}_{u+1,v}$ ($c^+_{i,j,u+1,v} = 1$); otherwise there is no constraint; these constraints express both the forwarding of communications and the stopping condition;
– there are similar constraints for other directions:
$\forall i, j, u > 1, v,\ c^N_{i,j,u,v} \leq c^+_{i,j,u-1,v} + \sum_k x_{j,k,u-1,v} \leq 2 - c^N_{i,j,u,v}$;
$\forall i, j, u, v < q,\ c^E_{i,j,u,v} \leq c^+_{i,j,u,v+1} + \sum_k x_{j,k,u,v+1} \leq 2 - c^E_{i,j,u,v}$;
$\forall i, j, u, v > 1,\ c^W_{i,j,u,v} \leq c^+_{i,j,u,v-1} + \sum_k x_{j,k,u,v-1} \leq 2 - c^W_{i,j,u,v}$.

A set of constraints express the fact that no cycle can occur in the communications:

– $\forall i, j, p{>}u{>}1, q{>}v{>}1,\ c^N_{i,j,u+1,v} + c^S_{i,j,u-1,v} + c^E_{i,j,u,v-1} + c^W_{i,j,u,v+1} \leq \sum_k x_{i,k,u,v}$;
– $\forall i, j, q > v > 1,\ c^N_{i,j,2,v} + c^E_{i,j,1,v-1} + c^W_{i,j,1,v+1} \leq \sum_k x_{i,k,1,v}$;
– $\forall i, j, q > v > 1,\ c^S_{i,j,p-1,v} + c^E_{i,j,p,v-1} + c^W_{i,j,p,v+1} \leq \sum_k x_{i,k,p,v}$;
– $\forall i, j, p > u > 1,\ c^N_{i,j,u+1,1} + c^S_{i,j,u-1,1} + c^W_{i,j,u,2} \leq \sum_k x_{i,k,u,1}$;

– $\forall i,j,p>u>1,\ c^N_{i,j,u+1,q} + c^S_{i,j,u-1,q} + c^E_{i,j,u,q-1} \le \sum_k x_{i,k,u,q}$;
– $\forall i,j,\ c^N_{i,j,2,1} + c^W_{i,j,1,2} \le \sum_k x_{i,k,1,1}$;
– $\forall i,j,\ c^S_{i,j,p-1,1} + c^W_{i,j,p,2} \le \sum_k x_{i,k,p,1}$;
– $\forall i,j,\ c^N_{i,j,2,q} + c^E_{i,j,1,q-1} \le \sum_k x_{i,k,1,q}$;
– $\forall i,j,\ c^S_{i,j,p-1,q} + c^E_{i,j,p,q-1} \le \sum_k x_{i,k,p,q}$.

Another constraint expresses the fact that the mapping is a DAG-partition:
– $\forall i,i',j,k,u,v,\ x_{i',k,u,v} \ge \ell^*_{i,i'} \times \ell^*_{i',j} \times (x_{i,k,u,v} + x_{j,k,u,v} - 1)$: if two stages S_i and S_j are mapped onto the same core $\mathcal{C}_{u,v}$, then any stage $S_{i'}$ which has an incoming dependency from S_i and an outgoing dependency from S_j must be mapped onto the same core, otherwise there would be a cycle in the partition.

Finally, we express the fact that the constraint on the period is fulfilled:
– $\forall u,v,k,\ \sum_i x_{i,k,u,v} \times w(i) \le T \times m_{k,u,v} \times s(k)$: constraint on computations;
– $\forall u,v\ \sum_{i,j} c^N_{i,j,u,v} \times \delta(i,j) \le T \times BW$: constraint on north communications;
– $\forall u,v\ \sum_{i,j} c^S_{i,j,u,v} \times \delta(i,j) \le T \times BW$: constraint on south communications;
– $\forall u,v\ \sum_{i,j} c^W_{i,j,u,v} \times \delta(i,j) \le T \times BW$: constraint on west communications;
– $\forall u,v\ \sum_{i,j} c^E_{i,j,u,v} \times \delta(i,j) \le T \times BW$: constraint on east communications.

Objective function

We aim at minimizing the energy consumption, which writes:

$$\min \begin{pmatrix} \sum_{u,v} \Big(\sum_k m_{k,u,v} \times Estat \\ \quad + \sum_{i,k} x_{i,k,u,v} \times w(i) \times Edyn(k) \Big) \\ \quad + \sum_{u,v,i,j} c^+_{i,j,u,v} \times \delta(i,j) \times E^{(\text{bit})} \end{pmatrix}.$$

The objective function is linear, as well as all the constraints. Since the variables are boolean, this is an integer linear program.

4.5 Heuristics

In this section, we describe the five heuristics that we have designed and implemented, thus providing practical solutions to the MINENERGY(T) problem. The first heuristic, **Random** (Section 4.5.1), performs a random mapping, and it is used for comparison purposes. Then we propose a greedy heuristic, **Greedy**, in Section 4.5.2, a heuristic based on a two-dimensional dynamic programming algorithm, **DPA2D**, in Section 4.5.3. Finally, we design two one-dimensional heuristics in Section 4.5.4: **DPA1D** builds upon the theoretical results of Section 4.4.1 and computes the optimal one-dimensional solution, while **DPA2D1D** computes the solution with the **DPA2D** heuristic, used in a one-dimensional setting.

4.5.1 Random heuristic

This first heuristic calls a procedure which works in two steps. The procedure first randomly builds a DAG-partition of the initial SPG, while ensuring that the period is matched for computations: we choose randomly a speed for the core which will handle the current subgraph G (initially, the source of the SPG), and we keep a list of stages of the SPGs that can be added to G while maintaining a DAG-partition. We pick a stage from this list randomly as long as computations do not exceed the period. When moving to the next core, we choose the first stage in the current list and iterate. In the second step,

we decide randomly on which core each subgraph is mapped, and communications are done following a XY routing: a communication from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v'}$ follows horizontal links from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v}$, and then vertical links from $\mathcal{C}_{u',v}$ to $\mathcal{C}_{u',v'}$. If the period is not exceeded on any communication link, then the mapping is valid, otherwise there is no solution.

For each problem instance, **Random** calls ten times this procedure, and keeps the solution which minimizes the energy consumption, if there is at least one valid solution; otherwise it fails.

4.5.2 Greedy heuristic

Given a speed $s \in \mathsf{S}$, this heuristic greedily assigns the SPG onto the platform, on which all cores are running at speed s. The greedy assignment is done through procedure **greedy**(s). The idea is to try all possible speed values, and to keep the best solution.

The greedy procedure **greedy**(s) works as follows: we keep a list of cores which are ready to be processed, and for each core, a list of successors, together with the corresponding outgoing communications. Initially, the only core in the list is $\mathcal{C}_{1,1}$, and we assign to this core the source stage S_1. The corresponding list of successors corresponds to the successors of S_1 in the SPG, and they are sorted by non-increasing communication volume to S_1.

When we process a core $\mathcal{C}_{u,v}$, we successively try to add some of the successors (from the current list) to this same core until the list is empty or the period is exceeded for computations on $\mathcal{C}_{u,v}$.

For each set of stages mapped onto $\mathcal{C}_{u,v}$ and the corresponding list of successors, we greedily share the corresponding communications between neighboring cores $\mathcal{C}_{u,v+1}$ and $\mathcal{C}_{u+1,v}$: communications are taken from the sorted list and assigned to the core which has currently the smallest amount of incoming communications. Then, we check that the partitioning is correct (no cycles in the dependence graph, i.e., we have a DAG-partition), and we check whether the bound on the period is achieved, both for computations and communications. If it is correct, we save the current solution before adding one more stage onto core $\mathcal{C}_{u,v}$ and iterating with one more stage on $\mathcal{C}_{u,v}$.

At the end of the iteration, we keep the last valid (saved) solution, i.e., the valid solution with the most number of stages onto $\mathcal{C}_{u,v}$. Cores $\mathcal{C}_{u,v+1}$ and $\mathcal{C}_{u+1,v}$ are then added to the list of ready cores, together with the list of successors (i.e., the stages that can either be assigned to this core, or forwarded to the neighboring cores).

The procedure finishes when the list of ready cores is empty, which means that all stages have been processed. Otherwise, the heuristic fails, and we move to the next speed. The energy for the mapping obtained with a given speed is computed by first *downgrading* the speed of each core, if possible: the procedure returns the mapping, and then we compute the amount of computations on each core, and set the core to the slowest possible speed, in order to save energy. Cores which are not used are turned off. Finally, the **Greedy** heuristic selects the mapping which corresponds to the lowest energy consumption.

4.5.3 2D dynamic programming algorithm

This heuristic, called **DPA2D**, starts by mapping the initial SPG onto a $x_{\max} \times y_{\max}$ grid, following the labels of the nodes (see Section 4.3.1). Then, this grid is mapped onto the CMP, thanks to a double nested dynamic programming algorithm.

First, we perform a dynamic programming algorithm to cut the grid into a set of columns, which are to be mapped onto a column of cores. Let $\mathcal{E}(m, v, D)$ be the optimal energy consumption to compute the first m levels of the SPG (i.e., all stages S_i with $x_i \leq m$), using v columns of cores, regardless of the outgoing communications. D is then the corresponding distribution of outgoing communications, i.e., a list of triplets (y, b, i), where y is the row from which communication is outgoing (i.e., the com-

munication is initiated by core $\mathcal{C}_{y,v}$), b is the amount of data, and S_i is the destination stage. We enforce these communications to go through $\mathcal{C}_{y,v+1}$, and then the communication will be redistributed to the destination core through vertical links. The solution is $\mathcal{E}(x_{\max}, q, D)$, and the recurrence is written as:

$$\mathcal{E}(m, v, D) = \min_{m'<m} \begin{pmatrix} \mathcal{E}(m', v-1, D') + \mathcal{E}^{\text{comm}}(D') \\ +\mathcal{E}^{\text{col}}(m'+1, m, D', D) \end{pmatrix},$$

with the initialization $\mathcal{E}(m, 1, D) = \mathcal{E}^{\text{col}}(1, m, \emptyset, D)$.

D' is the distribution of outgoing communications corresponding to the m' which leads to the optimal energy consumption, i.e., obtained with $\mathcal{E}(m', v-1, D')$.

$\mathcal{E}^{\text{comm}}(D')$ is the energy consumption induced by communications from column $v-1$ to column v (on horizontal links), given the distribution D' of outgoing communications of column $v+1$. If the bandwidth is exceeded on one of these horizontal links (i.e., $\exists 1 \leq y \leq p$ such that $\sum_{(y,b,i)\in D'} b > BW$), we set $\mathcal{E}^{\text{comm}}(D') = +\infty$.

$\mathcal{E}^{\text{col}}(m_1, m_2, D', D)$ is the optimal energy consumption of the column of the CMP which is processing stages S_i with $m_1 \leq x_i \leq m_2$: it accounts both for computations, and for vertical communications in the column, given the distribution of outgoing communications of the previous column, D'. The distribution of outgoing communications of this column is then D. Note that in the recurrence, D is an output of $\mathcal{E}^{\text{col}}(m'+1, m, D', D)$, while D' is an output of $\mathcal{E}(m', v-1, D')$. The values of $\mathcal{E}^{\text{col}}$ (and therefore, distribution D) are computed thanks to another dynamic programming algorithm: we compute $\mathcal{E}^{\text{col}}_{(m_1,m_2,D',D)}(g, u)$, which corresponds to the mapping of stages S_i, with $m_1 \leq x_i \leq m_2$ and $y_i \leq g$, onto the u first cores of a column of the CMP. As before, D' is an input, it corresponds to the distribution of outgoing communications arriving into the current column, while D is the distribution of outgoing communications of the current column for the solution which minimizes the energy consumption. Then we have $\mathcal{E}^{\text{col}}(m_1, m_2, D', D) = \mathcal{E}^{\text{col}}_{(m_1,m_2,D',D)}(y_{\max}, p)$.

For the distribution within a column, the recurrence writes:

$$\mathcal{E}^{\text{col}}_{(m_1,m_2,D',D)}(g, u) = \min_{g' \leq g} \begin{pmatrix} \mathcal{E}^{\text{col}}_{(m_1,m_2,D',D)}(g', u-1) \\ +\mathcal{E}^{\text{cal}}_{(m_1,m_2,D)}(g'+1, g) \\ +\mathcal{E}^{\text{ver}}_{(m_1,m_2,D')}(g'+1, g, u-1) \end{pmatrix},$$

with the initialization $\mathcal{E}^{\text{col}}_{(m_1,m_2,D',D)}(0, u) = 0$, and no outgoing communications from row 1 to row u, except the communications from D' that must be forwarded to the next column.

$\mathcal{E}^{\text{ver}}_{(m_1,m_2,D')}(g'+1, g, u-1)$ is the energy consumption of the vertical communications between cores $u-1$ and u in the column. These communications can either come from two dependent stages of the column, or be forwarded from the previous column (D'). If the bandwidth of the link is exceeded, we set the value to $+\infty$.

Finally, $\mathcal{E}^{\text{cal}}_{(m_1,m_2,D)}(g'+1, g)$ is the optimal energy consumption of a core which is computing all stages S_i such that $m_1 \leq x_i \leq m_2$, and $g'+1 \leq y_i \leq g$. If the period cannot be respected, or if the corresponding partition does not respect the DAG-partition constraint, the value is set to $+\infty$. Moreover, this function is adding to distribution D the communications from a stage S_i to another stage S_j, with $x_j > m_2$. These communications will occur on row u.

Note that in the recursive computation of $\mathcal{E}^{\text{col}}$, we can have $g' = g$, which means that no stage is assigned to core $\mathcal{C}_{u,v}$. This may happen if there are not enough stages in the column, or if this would save communications.

4.5.4 1D heuristics

The two last heuristics configure the CMP as a uni-directional uni-line CMP with $r = p \times q$ cores, by embedding it into the bi-directional platform as a *snake*:

$$\begin{array}{llll} \mathcal{C}_{1,1} & \rightarrow \mathcal{C}_{1,2} \rightarrow \cdots \rightarrow & & \mathcal{C}_{1,q} \\ & & & \downarrow \\ \mathcal{C}_{2,1} & \leftarrow \cdots \leftarrow \mathcal{C}_{2,q-1} \leftarrow & & \mathcal{C}_{2,q} \\ \downarrow & & & \\ \mathcal{C}_{3,1} & \rightarrow \mathcal{C}_{3,2} \rightarrow \ldots & & \end{array}$$

The **DPA1D** heuristic builds upon the theoretical results of Section 4.4, and computes the optimal solution of the dynamic programming algorithm of Theorem 4.1 with $r = p \times q$ cores. The mapping is then done along the snake; no other communication link is used.

Note that if the SPG is a linear chain, even if there are communication costs, then this heuristic is optimal, since any other solution could not exploit the communication links discarded with the snake structure. It is also optimal for any SPG without communication. However, **DPA1D** may take wrong decisions when communications are intensive, since it is restricted to a subset of communication links. Moreover, its complexity of $O(p \times q \times n \times n^{2y_{\max}})$ makes it intractable for SPGs with large $y_{\max}$.

Finally, the **DPA2D1D** heuristic computes the solution with the **DPA2D** heuristic (Section 4.5.3) on a $1 \times r$ CMP, and then do the mapping along the snake, similarly to **DPA1D**. The goal of this heuristic is to obtain efficient solutions when communications are not too intensive, and when the optimal **DPA1D** cannot find a solution in reasonable time.

4.6 Simulation results

This section reports simulation results assessing the performance of the various heuristics. As for the applications, we use both real-life applications taken from the *StreamIt* suite [109], and randomly generated applications, which allows us to cover a broader spectrum. As for the target platform, we use 4×4 and 6×6 CMP grids, whose hardware characteristics are representative of state-of-the-art devices. The source code for all simulations is publicly available at [102].

4.6.1 Simulation setting

Streaming applications

***StreamIt* suite.** There are 12 workflows in the *StreamIt* suite [109]. Their main characteristics are summarized in Table 4.1, where we give the size n, the maximum label values $y_{\max}$ and $x_{\max}$, and their *computation-to-communication ratio (CCR)*, defined as the sum $\sum_{i=1}^{n} w_i$ of all computations over the sum $\sum_{L_{i,j} \in \mathcal{E}} \delta_{i,j}$ of all communications. We observe in Table 4.1 that all workflows have a large CCR, hence are compute-intensive rather than data-intensive. In the simulations, we first use the workflows as such, with the original CCR values, and then we scale communication weights (the $\delta_{i,j}$) to change each CCR successively to 10, 1, and 0.1, so as to assess the impact of the communications on the performance of the heuristics.

Randomly generated. We randomly build SPG applications (by applying recursively series and parallel compositions of SPG applications), and we extract their size n, their elevation $y_{\max}$, together with their computation-to-communication ratio (CCR).

Index	Name	n	$y_{\max}$	$x_{\max}$	CCR
1	Beamformer	57	12	12	537
2	ChannelVocoder	55	17	8	453
3	Filterbank	85	16	14	535
4	FMRadio	43	12	12	330
5	Vocoder	114	17	32	38
6	BitonicSort	40	4	23	6
7	DCT	8	1	8	68
8	DES	53	3	45	7
9	FFT	17	1	17	17
10	MPEG2-noparser	23	5	18	9
11	Serpent	120	2	111	9
12	TDE	29	1	29	12

Table 4.1: Characteristics of the *StreamIt* workflows.

CMP configuration

For processor speeds and power consumption, we use the model of the Intel Xscale [60], following [31, 28, 89]. There are five speeds for each core:

$$s_{u,v} = (0.15, 0.4, 0.6, 0.8, 1)\ GHz,$$

with power consumption $P^{(\text{comp})}_{s_{u,v}} = (80, 170, 400, 900, 1600)\ mW$. We assume that the power consumption of the processor when it is idle is $P^{(\text{comp})}_{\text{leak}} = 80mW$. We use 16-byte wide communication links [93], whose bandwidths are $BW = 16 \times 1.2\ Gbytes$, which is reasonable according to [93]. Note that from the communication prospective, decreasing CCR has the same effect on the results as decreasing the width of the communication link below 16 bytes. The link energy is assumed to be between 1 and 10 picojoule per bit [25]; we fix $E^{(\text{bit})} = 6pJ$. Finally, we use $P^{(\text{comm})}_{\text{leak}} = 0$ without loss of generality (because for all heuristics the same quantity $P^{(\text{comm})}_{\text{leak}} \times T$ will be added to the total energy).

Period bound T

We need to find a meaningful value of T for each workflow. Indeed, if T is too large, all heuristics will map all stages onto a single processor running at the slowest speed, while if T is too small, all heuristics will fail. We choose T as follows: for each workflow, we start with $T = 1s$. With such a period, we observe that at least one heuristic succeeds. Then we iteratively divide the period by a factor of 10 and run all heuristics under this new value until all heuristics fail. We retain the period as the penultimate value, which is the last one before total failure. Note that this value depends upon the workflow, and that it is chosen to give some tightness to the mapping problem: at least one heuristic succeeds to find a mapping that matches the bound T, but none does for $T/10$.

4.6.2 Simulation results

StreamIt suite

In Figures 4.7 and 4.8, we plot the energy computed by the four heuristics for each application, given a CMP size (4×4 or 6×6) and a CCR ratio (set to the original value, 10, 1 and 0.1). On the

horizontal x axis, each group corresponds to an application, and x is the number of the application in Table 4.1. On the vertical axis, we plot the energy found by each heuristic, normalized by the minimum value obtained over all heuristics (so that the best heuristic returns 1, and the other ones return higher values). The **DPA1D** heuristic fails to return a solution for the first four applications, because there are too many possible splits to explore, and it is not plotted for those applications. More generally, each time a heuristic fails on a given application, it does not appear on the corresponding graph.

4×4 CMP grid. Results for a 4×4 CMP grid are given in Figure 4.7. When computations are predominant, i.e., when the CCR is set to its original value, or uniformly equal to 10, we observe that **Greedy**, **DPA2D**, **DPA1D** and **DPA2D1D** return similar results, and that **Random** always is within a factor of two. We also observe that **DPA2D** often fails on graphs with small elevation (linear graphs), because it wastes a lot of cores. For instance, if the application is exactly a pipeline (workflows numbered 7, 9 and 12), **DPA2D** can only enroll 4 cores over the 16 that are available. This fact holds true irrespective of the CCR.

When communications are more important, i.e., when the CCR is uniformly set to 1 or 0.1, **Random** gets much worse than the other heuristics: if it does not fail, its energy is between 2 and 4 times worse than the best one. In a general manner, we see that **DPA2D** is the best heuristic when the application graph has a high elevation.

We point out that **DPA1D** and **DPA2D1D** are the only successful heuristics for the workflow 11, whatever the CCR ratio is. This workflow fits very well with the main design idea of **DPA1D** and **DPA2D1D**: it is a pipeline-like graph (its elevation is only 2) with numerous stages. The other heuristics fail to find a good load-balance of computations and communications for this application.

The difference between **DPA1D** and **DPA2D1D** is tiny: when **DPA1D** finds a solution, **DPA2D1D** finds a close one, and there is only one graph (numbered 5) on which **DPA2D1D** suceeds, whereas **DPA1D** fails, because of the high memory complexity. Note that, in some cases, the solution of **DPA1D** is better than that of **DPA2D1D**, confirming that **DPA2D1D** does not return the optimal 1D mapping.

Altogether, **Greedy** seems to be a general-purpose heuristic that succeeds on most graphs, and it is always superior to **Random**. On the contrary, **DPA1D**, **DPA2D1D** and **DPA2D** are "specialized" heuristics, the first two heuristics are very efficient for long and almost linear graphs but not good for fat graphs of large elevation, and the last one behaving just as the opposite.

6×6 CMP grid. Results for a 6×6 CMP grid are given in Figure 4.8. Because the target grid is larger, it is easier to find a mapping that matches the period bound, especially for applications with a small number of stages. This is quantified in Table 4.2, where we report the number of failures for each heuristic.

We observe that the difference between solutions of **DPA2D1D** and solutions of **DPA1D** almost disappears. Otherwise, the conclusion remains more or less the same as on the 4×4 CMP grid, with **Greedy** always successful but also always inferior to one of the three specialized heuristics, **DPA1D**, **DPA2D1D** and **DPA2D**, depending upon the graph shape.

Platform size	**Random**	**Greedy**	**DPA2D**	**DPA1D**	**DPA2D1D**
4×4	5	4	16	20	16
6×6	0	0	17	20	8

Table 4.2: Number of failures for each heuristic (out of 48 instances per CMP grid size).

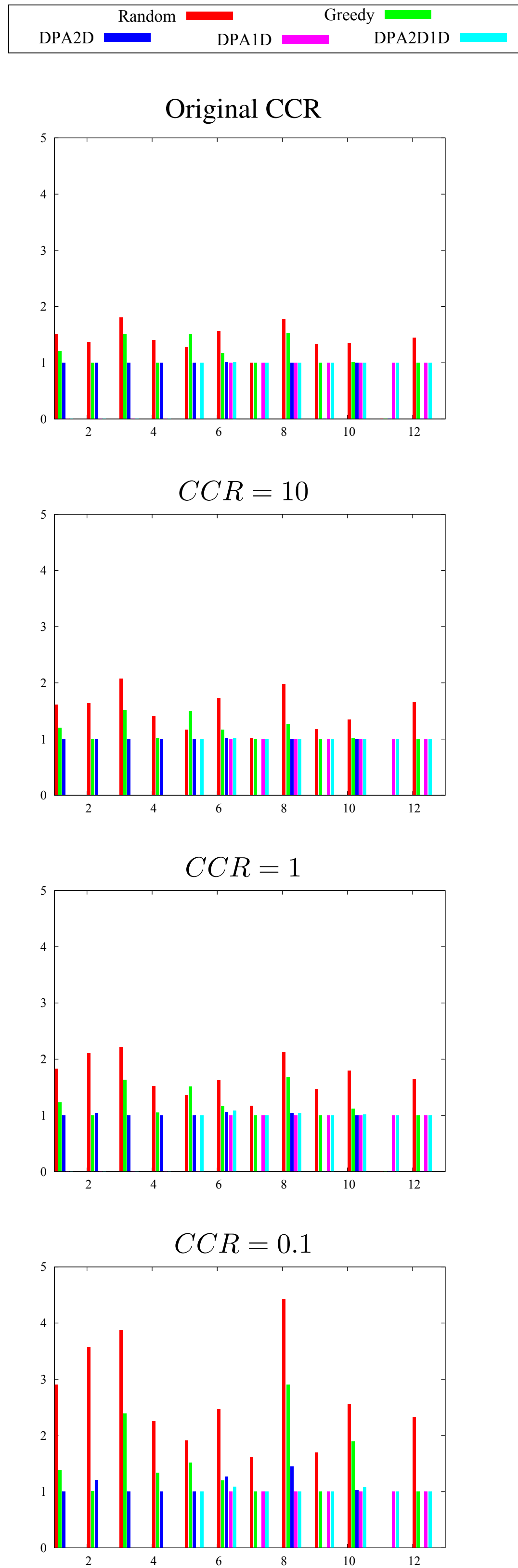


Figure 4.7: Normalized energy on the set of applications for a 4×4 CMP grid.

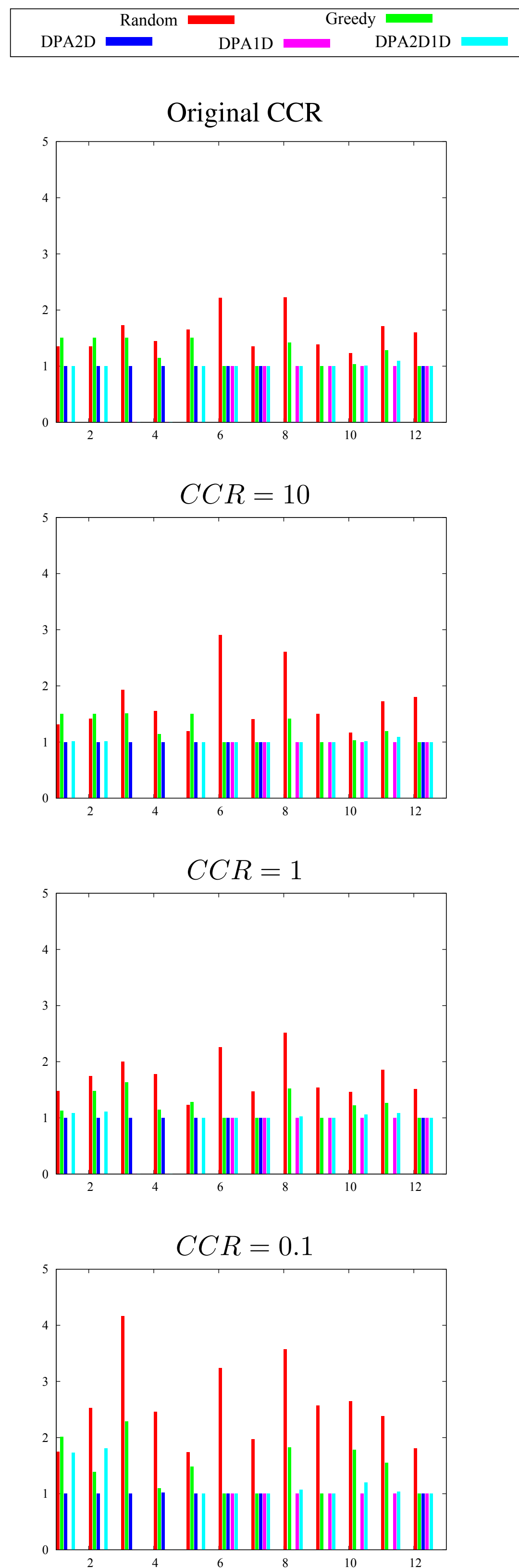


Figure 4.8: Normalized energy on the set of applications for a 6×6 CMP grid.

Random SPGs

For the randomly generated SPGs, we plot four sets of three graphs; in each set, the three graphs are obtained for a given CCR (10, 1 or 0.1), whereas each set corresponds to a value of the couple (n, p), where the number of nodes n can be 50 or 150 and the number of cores p in a row of the square CMP can be 4 or 6.

On the horizontal axis, we represent the elevation of the SPG. For each value of the elevation, we average the results obtained on 100 randomly generated applications. On the vertical axis, we plot the inverse of the energy found by each heuristic, normalized to the minimum value obtained over all heuristics (so that the best heuristic returns 1, and the other ones return smaller values).

With 50 nodes and a 4×4 CMP grid. Results are given in Figure 4.9. When computations are predominant, i.e., when the CCR is uniformly equal to 10, we observe that the two 1D heuristics always return good results. For small elevations, **DPA1D** is the best, but it often fails as soon as the elevation is greater than 4, thus leading to poor results. **DPA2D1D** returns very good results whatever the elevation of the graph. The 2D heuristic **DPA2D** is the best for elevations greater than 6, but it often fails on graphs with small elevation, because it wastes a lot of cores. For instance, if the application is exactly a pipeline (elevation 1), **DPA2D** can only enroll 4 cores over the 16 that are available. This fact holds true irrespective of the CCR. **Greedy** and **Random** are not as good, but **Greedy** always outperforms **Random**.

When communications and computations are more balanced (CCR of 1), similar results can be observed, but **DPA2D1D** is a bit further from the best solution, since it cannot utilize all the communication links. Finally, for communication-intensive applications (CCR of 0.1), **Random** gets much worse than the other heuristics: its energy can be up to 10 times worse than the best one. Also, the 1D heuristics do not perform well, except for small elevation graphs, because of their restriction in the communication pattern. In a general manner, we see that **DPA2D** is the best heuristic when the application graph has a high elevation.

Number of failures. In Table 4.3, we report the number of failures for each heuristic, again with 50 nodes and a 4×4 CMP grid. With a large CCR (10 or 1), **DPA2D1D** almost always succeeds to find a solution, which are in turn pretty good (see Figure 4.9). **Greedy** is always reasonably robust, whatever the CCR, and is followed closely by **Random**. **DPA2D** fails a bit more frequently because it does not often succeed with graphs of small elevation, as explained earlier. Finally, **DPA1D** succeeds only for graphs of small elevation, which leads to a very high failure rate.

Other results. We have performed further simulations on larger applications and/or different CMP grid sizes, see Figures 4.10, 4.11, and 4.12. Overall, the conclusions remain the same, and they confirm the results derived from the real-life *StreamIt* applications.

CCR	**Random**	**Greedy**	**DPA2D**	**DPA1D**	**DPA2D1D**
10	58	56	156	1516	2
1	58	56	156	1520	4
0.1	300	287	348	1340	916

Table 4.3: Number of failures (out of 2000 instances per CCR value).

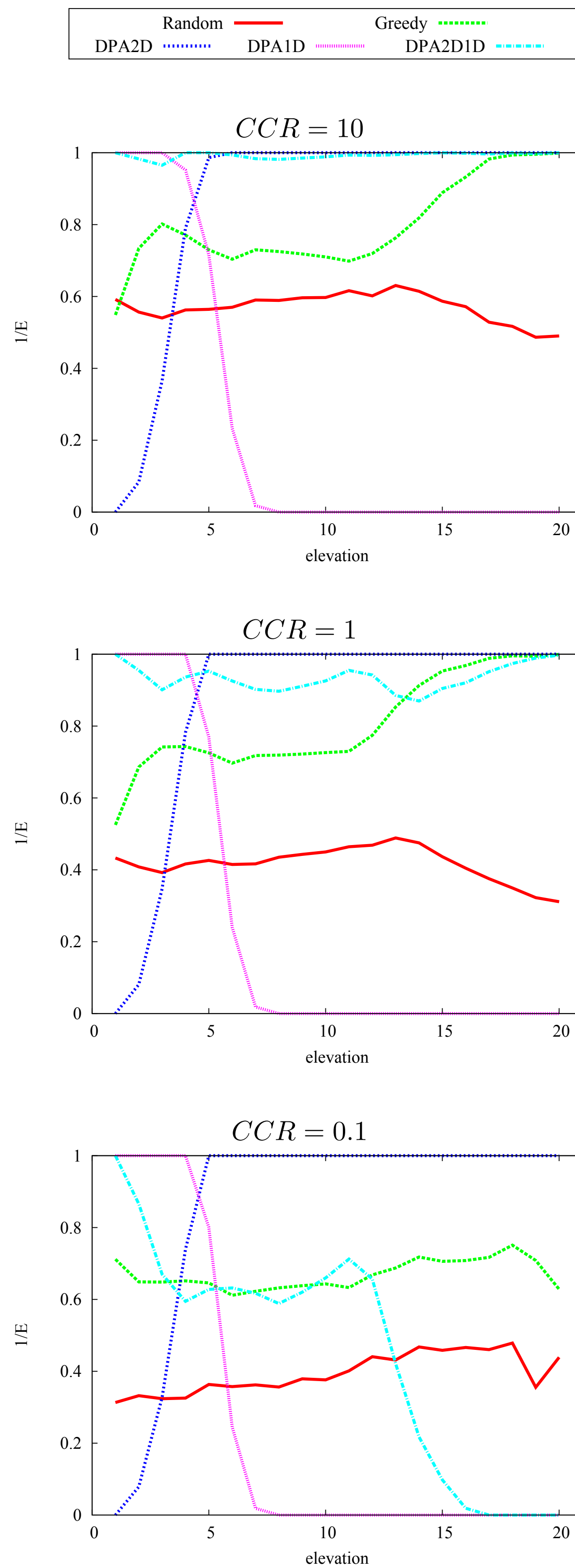


Figure 4.9: Normalized energy inverse on a random set of applications of 50 nodes for a 4×4 CMP.

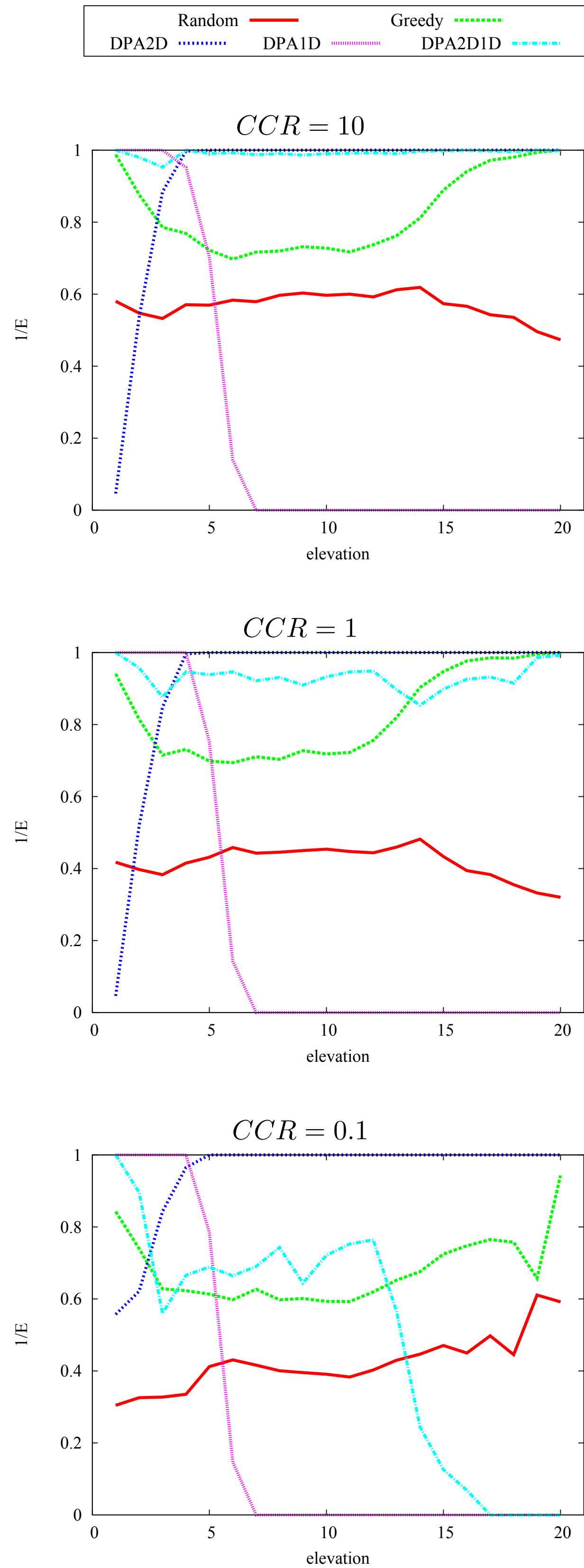


Figure 4.10: Normalized energy inverse on a random set of applications of 50 nodes for a 6×6 CMP.

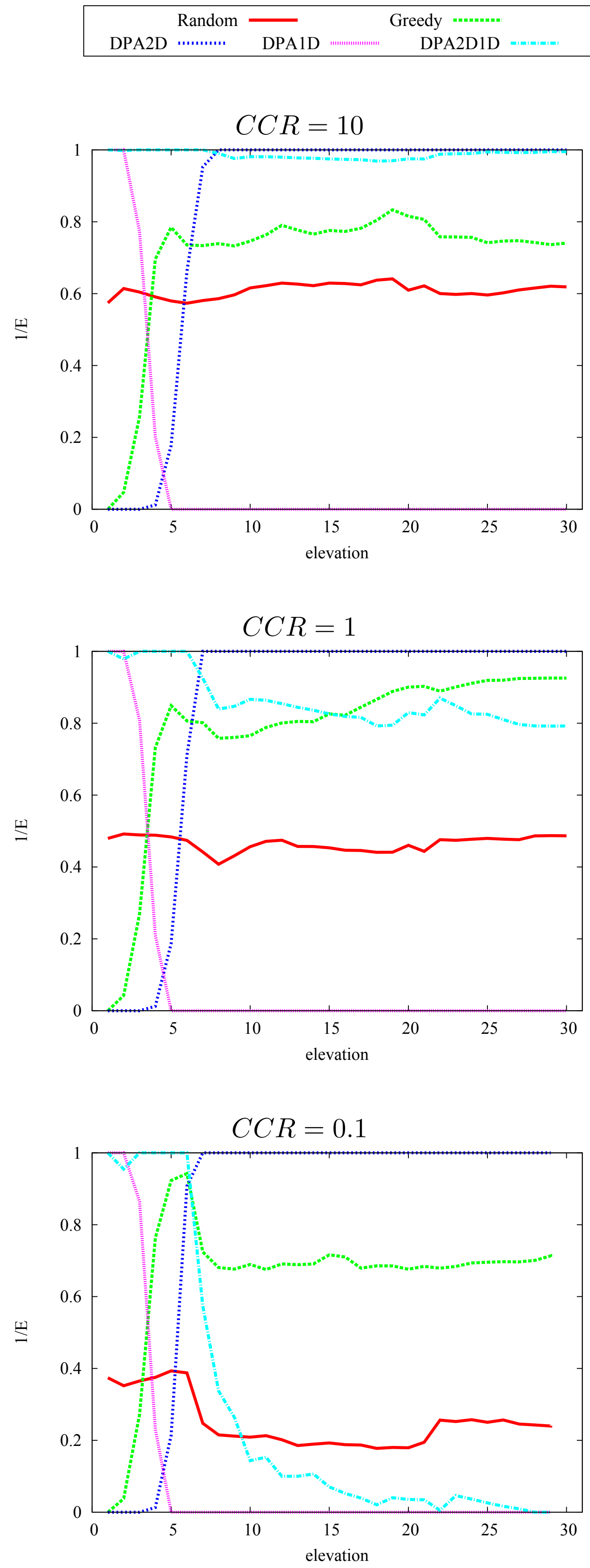


Figure 4.11: Normalized energy inverse on a random set of applications of 150 nodes for a 4×4 CMP.

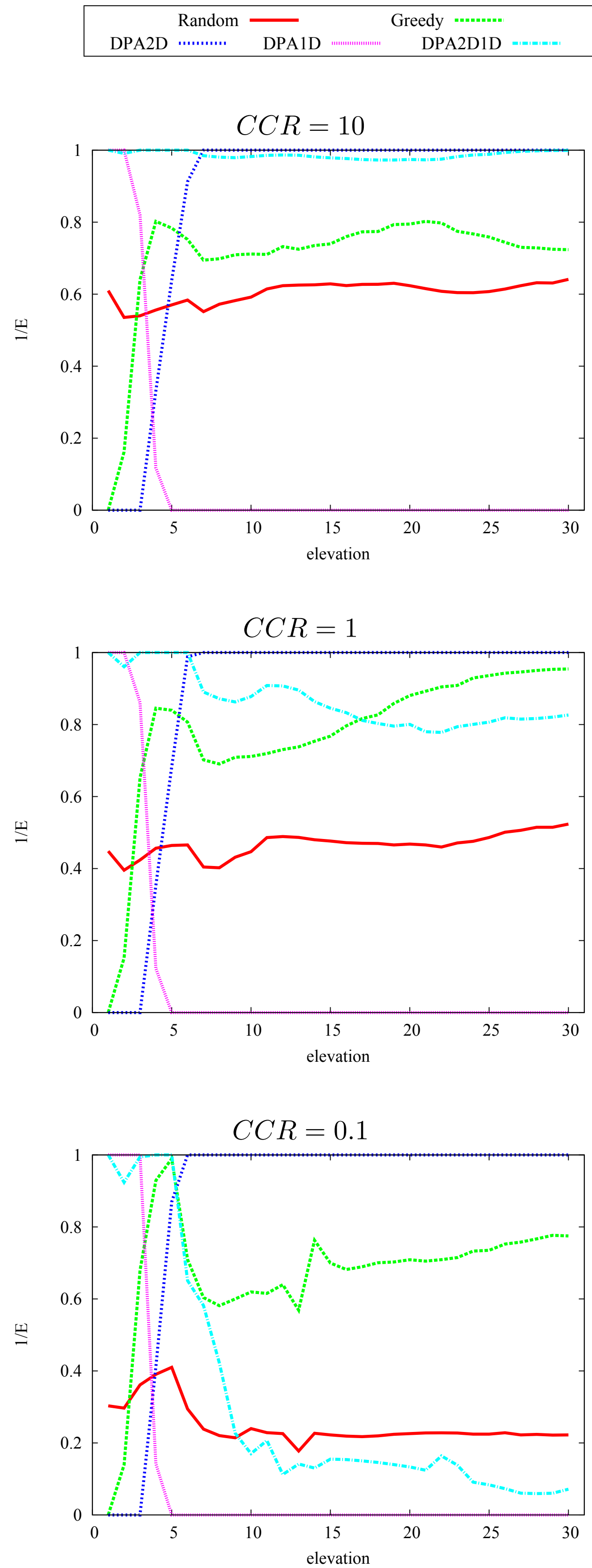


Figure 4.12: Normalized energy inverse on a random set of applications of 150 nodes for a 6×6 CMP.

4.7 Conclusion

This chapter contributes to the efficient utilization of multicores by considering an important class of streaming applications that can be modeled by a series-parallel graph, and studying the problem of mapping these applications to 2-dimensional tiled CMP architectures. The objective of the mapping is to minimize the energy consumption while maintaining a given level of performance, reflected by the rate of processing the data streams. Both processing and communication capabilities and power consumption are considered during the mapping, but it is assumed that only the processing power can be managed through dynamic voltage and frequency scaling. We will consider systems in which the communication power can also be managed in future work.

From a theoretical angle, we showed that most of the bi-criteria mapping problems were NP-complete, with the notable exception of uni-directional uni-line CMPs, for which an elaborated dynamic programming algorithm returns the optimal solution. The latter result holds true only for bounded-elevation SPGs, and the problem becomes NP-complete otherwise, which provides yet another evidence of the interest to restrict to particular graph structures rather than to deal with arbitrary DAGs. We strongly believe that bounded-elevation SPGs represent a very interesting trade-off, as they combine a large practical significance while being amenable to rigorous analysis.

From a practical angle, the simulations conducted with the *StreamIt* suite [109] and the randomly generated SPGs confirmed the efficiency of the main design principles underlying the various heuristics. While **Greedy** is the most robust approach, it is always superseded by at least one of the three specialized algorithms, **DPA1D** for long pipeline-like graphs, **DPA2D** for fat graphs of large elevation or **DPA2D1D** for any graph containing low communication weights and for graphs of low elevation.

Finally, our future research will investigate general mappings, and assess the difference with DAG-partition mappings, both from a theoretical and a practical perspective. We also hope to succeed in simplifying the integer linear program for some problem instances, thereby providing an absolute measure of the quality of the various heuristics.

Chapter 5

Manhattan routing on chip multiprocessors

5.1 Introduction

In the previous chapter, we designed several heuristics that map streaming applications onto a chip multiprocessor (CMP). Although the energy consumed by the communications was taken into account in the model and in the decisions of the heuristics, we have been mainly focused on the minimization of the energy consumed by the computations. This was motivated by the fact that the very most of current power consumption of a processor takes place in the CPU. However, advances in technology enables the integration of ever-increasing numbers of processor cores into a single CMP [21], and this integration creates the need for high bandwidth on-chip communication. It also increases the power consumption of a CMP and necessitates the use of clever management technique to reduce power consumption and mitigate its effect on chip temperature and reliability. A significant fraction of the CMP power will be consumed in the on-chip interconnection [93, 56] and we need to reduce and manage this power.

In this chapter, we consider CMPs with mesh interconnections and we investigate the reduction of the power consumed for on-chip communication through power-aware routing. Specifically, we consider the following problem: given a set of inter-node communications on the CMP, each with some bandwidth requirement expressed in bytes per second, find the best routes for these communications so that the total power consumed on all the communication links is minimized. Here we target the problem at the system level rather than at the application level: there are several parallel applications executing on the CMP, and each of them has been mapped onto a set of nodes, resulting in one or several communications between CMP nodes. From a system's point of view, a communication between two nodes is characterized by its requested bandwidth (in terms of bytes per second) irrespective of the application that generates the communication.

Each communication is routed from source to destination along a given path using either source routing or table-based routing. The total power consumed for the communication consists of a static part (mostly resulting from leakage) and a dynamic part (which depends on the number of bytes transmitted). An effective technique for managing the power consumption of interconnection networks is based on scaling the frequency and voltage of the communication links to match the traffic traversing those links [105]. Specifically, assume that routing the communications is such that the total traffic on a link L_ℓ resulting from all communication is D_ℓ bytes per second. Hence, to satisfy the requests and minimize power consumption, link L_ℓ must operate at a frequency f_ℓ that matches or exceeds D_ℓ/W, where W is the width of the communication link in bytes. This translates into $f_\ell = D_\ell/W$ if we have a model with continuous frequencies, or into $f_\ell = f_{\min} \geq D_\ell/W$ if frequencies are discrete, where $f_{\min}$ is the lowest frequency matching the constraint. The dynamic power dissipated by link L_ℓ is proportional to the αth power of f_ℓ, where α is between 2 and 3. The total dynamic power dissipated by the

communications is the sum over all links.

The most natural and widely used algorithm to handle communications in 2-dimensional meshes is XY-routing: for each communication, data is first forwarded horizontally, and then vertically, from source to destination. However, many alternate routing paths can be used in meshes. In fact, all Manhattan paths from the source to the destination are natural candidates to route the message. This freedom in routing can help dramatically reduce power consumption, when the static part of the power consumption can be neglected. For example, if there are two equal-volume communications from the same source to the same destination, the first can be routed along an XY path and the second along a YX path, thus reducing the constraint on each link by half, and thereby reducing the power consumed on that link by a factor of 2^{α}; this reduces the total dynamic power consumption by $2^{\alpha-1}$. However, the number of links used is doubled in this case, and the static power consumption is doubled too. In the general case, given a set of communications, our goal is to determine one or several routing paths for each communication, so that the total power consumption is minimized. This requires that our heuristics achieve good trade-offs between static and dynamic power consumption. Note that we consider only shortest path (Manhattan) routing and we assume that a deadlock avoidance technique is used (such as resource ordering [52] or escape channels [42]).

The rest of the chapter is organized as follows. In Section 5.2 we survey related work in the domain of routing in CMPs. Then in Section 5.3, we expose the framework in which our results take place. The theoretical results (worst case analysis and NP-completeness) are presented in Section 5.4. Finally we describe the heuristics in Section 5.5, and show their performance in Section 5.6. We conclude in Section 5.7.

5.2 Related work

Routing algorithms for on-chip networks can be oblivious to the application traffic [104] or can dynamically adapt to that traffic [47]. If, however, the characteristics of the traffic are statically known, then routing algorithms can take advantage of that knowledge to optimize the performance of the interconnection network. For on-chip routing, there have been many proposals to design traffic-aware routes with the goal of maximizing the communication bandwidth and/or minimizing its delay [85, 72].

When power consumption of the network was recognized as a major component of the total power consumption in CMPs, many techniques have been investigated to manage the power on the links and switches of the interconnection network. Dynamic Voltage and Frequency Scaling (DVFS) and turning off unused links are among the most efficient techniques that can take advantage of the variation in traffic to reduce power [105, 4, 77]. Static knowledge of the traffic patterns obtained by compiler analysis was also used to optimize the frequency/voltage scaling of the individual interconnection links in the network [80]. Recent research proposes the adaptive use of back-gate biasing for managing the dynamic power of on-chip interconnect [75] and the dynamic redistribution of the power between the on-chip cores and routers to adapt to the variation in the computation and communication demands of applications [81].

In [106], an off-line link speed assignment algorithm was presented for energy efficient on-chip networks with voltage scalable links. Given the task graph of a periodic real-time application, genetic algorithms are used to first assign the tasks to processors and then to assign appropriate communication speeds to the communication links with the goal of reducing power consumption. In this chapter, we isolate the routing problem and provide theoretical results about its complexity. We also explore a number of heuristics to solve it in polynomial time.

5.3 Framework

In this section, we first describe the platform and power consumption model (Section 5.3.1). Then we formalize the communications that need to be routed (Section 5.3.2), and we discuss routing rules (Section 5.3.3). We are then ready to formally define the optimization problem (Section 5.3.4). Finally, we provide a brief comparison of the routing rules in Section 5.3.5.

5.3.1 Platform and power consumption model

The target platform is a CMP (**C**hip **M**ulti**P**rocessor), composed of $p \times q$ homogeneous cores $\mathcal{C}_{u,v}$, with $1 \leq u \leq p$, $1 \leq v \leq q$, arranged along a rectangular grid. There are two unidirectional opposite links between neighbor cores. Hence, vertically, for each $(u,v) \in \{1,\dots,p-1\} \times \{1,\dots,q\}$, there is a link $\mathcal{L}_{(u,v)\to(u+1,v)}$ from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u+1,v}$ and a link $\mathcal{L}_{(u+1,v)\to(u,v)}$ from $\mathcal{C}_{u+1,v}$ to $\mathcal{C}_{u,v}$. Similarly, horizontally, for each $(u,v) \in \{1,\dots,p\} \times \{1,\dots,q-1\}$, there is a link $\mathcal{L}_{(u,v)\to(u,v+1)}$ from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u,v+1}$ and a link $\mathcal{L}_{(u,v+1)\to(u,v)}$ from $\mathcal{C}_{u,v+1}$ to $\mathcal{C}_{u,v}$.

Let $succ_{u,v}$ be the set of destination cores of the outgoing links of $\mathcal{C}_{u,v}$ (i.e., the neighbor cores). Each link has a maximum bandwidth BW but is scalable: we can choose the fraction $f_{(u,v)\to(u',v')}$ of the bandwidth of the link from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v'} \in succ_{u,v}$ that is active. This means that $f_{(u,v)\to(u',v')} \times BW$ bytes can go from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v'}$ during one second, where $0 \leq f_{(u,v)\to(u',v')} \leq 1$.

We define the set of the active links $\mathcal{A}$ such that

$$\forall (u,v) \in \{1,\dots,p\} \times \{1,\dots,q\}, \quad \forall \mathcal{C}_{u',v'} \in succ_{u,v},$$
$$\mathcal{L}_{(u,v)\to(u',v')} \in \mathcal{A} \;\Leftrightarrow\; f_{(u,v)\to(u',v')} \neq 0.$$

We model the power consumption of the platform as the sum of a static part (the leakage power), and a dynamic part. The leakage power $P_{\text{leak}}^{(\text{comm})}$ is the power consumption of a router that is switched on, while the dynamic power depends on the active bandwidth of the link. More precisely,

$$P_{\text{dyn}}(\mathcal{L}_{(u,v)\to(u',v')}) = P_0 \times \left(f_{(u,v)\to(u',v')} BW\right)^{\alpha},$$

where P_0 is a constant and $2 < \alpha \leq 3$ [69].

Hence, if $\mathcal{L}_{(u,v)\to(u',v')} \in \mathcal{A}$, the power dissipated to send communications through $\mathcal{L}_{(u,v)\to(u',v')}$ is

$$P_{(u,v)\to(u',v')} = P_{\text{leak}}^{(\text{comm})} + P_0 \times \left(f_{(u,v)\to(u',v')} BW\right)^{\alpha}.$$

If $\mathcal{L}_{(u,v)\to(u',v')}$ is inactive, then $P_{(u,v)\to(u',v')} = 0$.

5.3.2 Communications

Since there is no distinction between the applications, we do not have to take care of which application a communication belongs to. And as the mapping of the applications is fixed, the communications can be viewed as follows. We are given a set $\{\gamma_1, \gamma_2, \dots, \gamma_{n_c}\}$ of n_c different communications; a communication is defined by $\gamma_i = (\mathcal{C}_{usrc(i),vsrc(i)}, \mathcal{C}_{usnk(i),vsnk(i)}, \delta_i)$, where $\mathcal{C}_{usrc(i),vsrc(i)}$ is the source core, $\mathcal{C}_{usnk(i),vsnk(i)}$ is the destination (sink) core, and δ_i is the number of bytes per second required by the message.

The routing of each communication γ_i is described as a path, denoted $path_i$. This path, of length ℓ_i, is a sequence of communication links

$$\left(\mathcal{L}_{(us_1,vs_1)\to(ud_1,vd_1)}, \;\dots,\; \mathcal{L}_{(us_{\ell_i},vs_{\ell_i})\to(ud_{\ell_i},vd_{\ell_i})}\right),$$

such that $\mathcal{C}_{us_1,vs_1} = \mathcal{C}_{usrc(i),vsrc(i)}$, $\mathcal{C}_{ud_{\ell_i},vd_{\ell_i}} = \mathcal{C}_{usnk(i),vsnk(i)}$, and for all $\ell \in \{1,\dots,\ell_i - 1\}$, $\mathcal{C}_{ud_\ell,vd_\ell} = \mathcal{C}_{us_{\ell+1},vs_{\ell+1}}$.

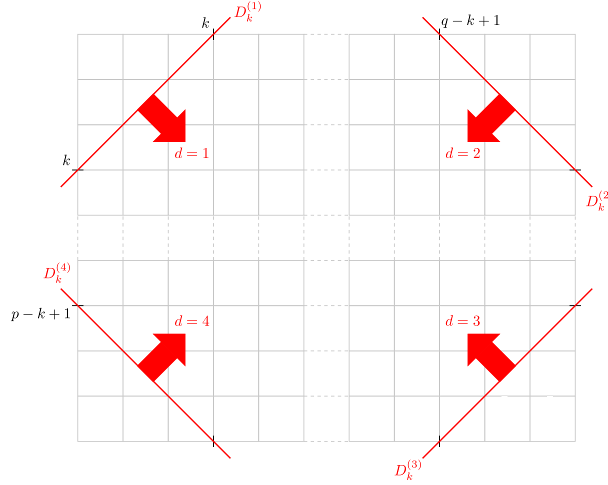


Figure 5.1: Location of the communications.

5.3.3 Routing rules

As stated and motivated earlier, we restrict the study to Manhattan paths, hence to shortest paths. Therefore, the length of any path for communication γ_i between $\mathcal{C}_{usrc(i),vsrc(i)}$ and $\mathcal{C}_{usnk(i),vsnk(i)}$ is $\ell_i = |usrc(i) - usnk(i)| + |vsrc(i) - vsnk(i)|$.

We define diagonals of cores $D_k^{(d)}$ (as illustrated in Figure 5.1) for all values of $k \in \{1, \ldots, q+p-1\}$, and for $d \in \{1, 2, 3, 4\}$:
- $\mathcal{C}_{u,v} \in D_k^{(1)} \Leftrightarrow u + v - 1 = k$;
- $\mathcal{C}_{u,v} \in D_k^{(2)} \Leftrightarrow u + q - v = k$;
- $\mathcal{C}_{u,v} \in D_k^{(3)} \Leftrightarrow p - u + q - v + 1 = k$;
- $\mathcal{C}_{u,v} \in D_k^{(4)} \Leftrightarrow p - u + v = k$.

Note that each core is in exactly four diagonals (one for each value of d). The index d corresponds to the *direction* of the diagonal.

We also define the direction d_i of communication γ_i, and the index $ksrc(i)$ of the diagonal of direction d_i that $\mathcal{C}_{usrc(i),vsrc(i)}$ belongs to (i.e., $\mathcal{C}_{usrc(i),vsrc(i)} \in D_{ksrc(i)}^{(d_i)}$), as:
- if $usrc(i) \leq usnk(i)$ and $vsrc(i) \leq vsnk(i)$, then $d_i = 1$ and $ksrc(i) = usrc(i) + vsrc(i) - 1$;
- if $usrc(i) \leq usnk(i)$ and $vsrc(i) > vsnk(i)$, then $d_i = 2$ and $ksrc(i) = usrc(i) + q - vsrc(i)$;
- if $usrc(i) > usnk(i)$ and $vsrc(i) > vsnk(i)$, then $d_i = 3$ and $ksrc(i) = p - usrc(i) + q - vsrc(i) + 1$;
- if $usrc(i) > usnk(i)$ and $vsrc(i) \leq vsnk(i)$, then $d_i = 4$ and $ksrc(i) = p - usrc(i) + vsrc(i)$.

With those definitions, since the paths are shortest paths, communications always move along the same direction. Formally, the ℓth communication link of $path_i$ goes from a core in $D_{ksrc(i)+\ell-1}^{(d_i)}$ to a core in $D_{ksrc(i)+\ell}^{(d_i)}$. Therefore, the index $ksnk(i)$ of the diagonal of direction d_i that $\mathcal{C}_{usnk(i),vsnk(i)}$ belongs to is $ksnk(i) = ksrc(i) + \ell_i$, i.e., $\mathcal{C}_{usnk(i),vsnk(i)} \in D_{ksrc(i)+\ell_i}^{(d_i)}$.

We are now ready to describe the different routing rules:

- *XY routing* (XY). Each communication goes horizontally first, then vertically.
- *Single-path Manhattan routing* (1-MP). The communication can take any path as described above.

• *s-paths Manhattan routing* (s-MP). A communication γ_i can be split into $s' \leq s$ distinct communications $\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,s'}$, of sizes $\delta_{i,1}, \delta_{i,2}, \ldots, \delta_{i,s'}$, where:

1. for each $s'' \in \{1, \ldots, s'\}$, $\quad \gamma_{i,s''} = (\mathcal{C}_{usrc(i),vsrc(i)}, \mathcal{C}_{usnk(i),vsnk(i)}, \delta_{i,s''})$;
2. $\sum_{s''=1}^{s'} \delta_{i,s''} = \delta_i$.

Note that for each $i \in \{1, \ldots, n_c\}$, since all $\gamma_{i,j}$ (for $j \in \{1, \ldots, s\}$) have the same source core and sink core, they all have the same length ℓ_i and direction d_i. However, since communications have been split, we can now choose different paths for each part of the former communications.

• *max-paths Manhattan routing* (max-MP). This is a special case of s-MP where the number of paths is not bounded, i.e., a communication can be split into any number of distinct communications. We bound this number in Section 5.4.

5.3.4 Problem definition

We are given a CMP, a set of communications $\{\gamma_1, \ldots, \gamma_{n_c}\}$, and a routing rule (XY or s-MP), with a maximum number s of paths for a single communication. A routing is defined by:

- for each $i \in \{1, \ldots, n_c\}$, a splitting into $\{\gamma_{i,1}, \ldots, \gamma_{i,s}\}$ if $s > 1$, otherwise $\gamma_{i,1} = \gamma_i$ for XY or 1-MP;
- for each $j \in \{1, \ldots, s\}$, the path $path_{i,j}$ of $\gamma_{i,j}$;
- for all $(u, v) \in \{1, \ldots, p\} \times \{1, \ldots, q\}$ and $\mathcal{C}_{u',v'} \in succ_{u,v}$, the fraction of bandwidth $f_{(u,v)\to(u',v')}$ used for the communication from $\mathcal{C}_{u,v}$ to $\mathcal{C}_{u',v'}$.

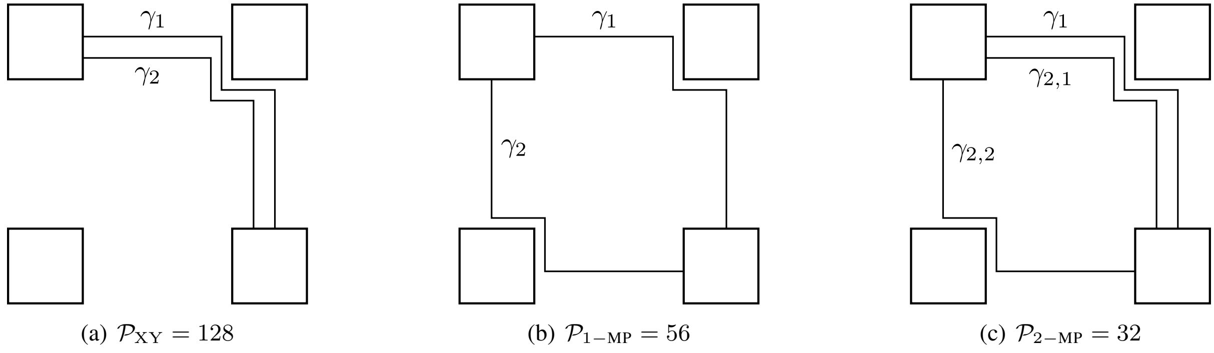
Our goal is to find a routing that minimizes the total power consumption, while ensuring that link bandwidths are not exceeded. This last constraint adds the volume of communication going through each link and checks that the fraction of bandwidth available is not exceeded: for all $(u, v) \in \{1, \ldots, p\} \times \{1, \ldots, q\}$ and $\mathcal{C}_{u',v'} \in succ_{u,v}$,

$$\sum_{\substack{i \in \{1,\ldots,n_c\}, j \in \{1,\ldots,s\} \\ \mathcal{L}_{(u,v)\to(u',v')} \in path_{i,j}}} \delta_{i,j} \quad \leq \quad f_{(u,v)\to(u',v')} \times BW.$$

5.3.5 Comparison of routing rules

Note first that XY routing is a restriction of 1-MP routing, which is itself a restriction of s-MP routing.

We give here an example such that there exists a 1-MP routing that is better than the XY routing, and there exists a s-MP routing that is better than any 1-MP routing. We set $P_{\text{leak}}^{(\text{comm})} = 0$, $P_0 = 1$, $\alpha = 3$, $BW = 4$, and we consider two communications $\gamma_1 = (\mathcal{C}_{1,1}, \mathcal{C}_{2,2}, 1)$ and $\gamma_2 = (\mathcal{C}_{1,1}, \mathcal{C}_{2,2}, 3)$. The XY routing is shown in Figure 5.2(a), and it leads to a power $\mathcal{P}_{\text{XY}} = 2 \times 4^3 = 128$. The best 1-MP routing is depicted in Figure 5.2(b), and leads to a power $\mathcal{P}_{1-\text{MP}} = 2 \times (1^3 + 3^3) = 56$. In the best 2-MP routing, γ_2 is split into $\gamma_{2,1} = (\mathcal{C}_{1,1}, \mathcal{C}_{2,2}, 1)$ and $\gamma_{2,2} = (\mathcal{C}_{1,1}, \mathcal{C}_{2,2}, 2)$ (see Figure 5.2(c)). The consumed power is then $\mathcal{P}_{2-\text{MP}} = 2 \times (2^3 + 2^3) = 32$.



(a) $\mathcal{P}_{\rm XY} = 128$ (b) $\mathcal{P}_{1-\rm MP} = 56$ (c) $\mathcal{P}_{2-\rm MP} = 32$

Figure 5.2: Comparison of routing rules.

5.4 Theoretical results

In this section, we first show (Section 5.4.1) how much power we can save if Manhattan routing can be used instead of XY routing. Then, we prove the NP-completeness of the problem of finding a Manhattan routing in Section 5.4.2.

5.4.1 Manhattan vs XY

Throughout this section we let $P_{\rm leak}^{(\rm comm)} = 0$ and $P_0 = 1$, so that routing policies aim at load-balancing communications as well as possible on all communication links. This scenario corresponds to communication-intensive applications: as the total communication volume increases, the dynamic part of the power consumption becomes more and more predominant. Note that if $P_{\rm leak}^{(\rm comm)}$ is very large and P_0 very small, then the problem becomes completely different, since the objective would be to group many communications on the same links, in order to minimize the total number of links that would be used in the end.

We start by counting the number of Manhattan paths going from $\mathcal{C}_{1,1}$ to $\mathcal{C}_{p,q}$, hence enabling us to characterize the maximum number of paths that can be used by a max-MP routing.

Lemma 5.1. *There are $\binom{p+q-2}{p-1}$ Manhattan paths going from $\mathcal{C}_{1,1}$ to $\mathcal{C}_{p,q}$.*

Proof. Any Manhattan path going from $\mathcal{C}_{1,1}$ to $\mathcal{C}_{p,q}$ uses exactly $p+q-2$ communication links, and the different paths are obtained by choosing which of these links are the $p-1$ vertical ones. ■

Single source and single destination. We start the comparison with communications that share the same source core and the same destination core. We study the worst case of an XY routing versus a multi-path Manhattan routing, in which the maximum number of communications is the number of different paths in the processor. This corresponds to the max-MP routing rule.

Theorem 5.1. *Given a $p \times q$ CMP with $q \geq p$, $q = O(p)$, and a set of communications to be routed from $\mathcal{C}_{1,1}$ to $\mathcal{C}_{p,q}$, the minimum upper bound for the ratio of the power consumed by an XY routing ($\mathcal{P}_{\rm XY}$) over the power consumed by a max-*MP *routing ($\mathcal{P}_{\max}$) is in $O(p)$.*

Note that the result holds true for a $p \times p$ square CMP as a particular case. Note also that it holds for the symmetric case of CMP with $p \geq q$ and $p = O(q)$, with a minimum upper bound in $O(q)$.

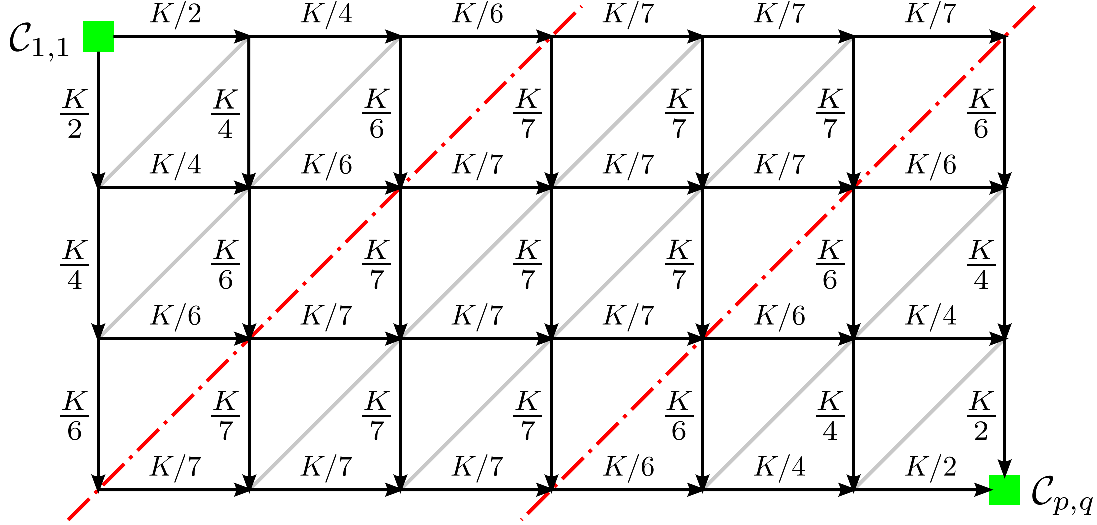


Figure 5.3: Ideal sharing of one communication.

Proof. We first prove that an upper bound of $\mathcal{P}_{\text{XY}}/\mathcal{P}_{\max}$ is in $O(p)$. Then, we show that this bound can indeed be achieved.

Let K be the total size of the communications to route (that is to say $K = \sum_{i\in\{1,\dots,n_c\}} \delta_i$). The XY routing is forwarding all these communications along the same route, leading to a power consumption $\mathcal{P}_{\text{XY}} = (p+q) \times K^\alpha$, and therefore $\mathcal{P}_{\text{XY}}$ is in $O(p \times K^\alpha)$ (recall that $q = O(p)$).

All communications, even if split in multiple paths (as allowed with a max-MP routing), follow the same diagonals in direction 1. For each $k \in \{1,\dots,q+p-2\}$, we let by $K_k^{(1)}$ be the sum of the γ_i for all $i \in \{1,\dots,n_c\}$ such that $ksrc(i) \leq k$ and $ksnk(i) > k$. Since all communications have the same source and destination, $K_k^{(1)} = K$ for each k. For a given $K_k^{(1)}$, the ideal way to map those communications is to distribute them among all the communication links from $D_k^{(1)}$ to $D_{k+1}^{(1)}$ (see Figure 5.3). Such a splitting cannot be achieved but provides a bound on how to load-balance the communication across the links. We have:

$$\mathcal{P}_{\max} \geq \sum_{k=1}^{p-1} 2k \left(\frac{K_k^{(1)}}{2k}\right)^\alpha + \sum_{k=p}^{q-1} (2p-1) \left(\frac{K_k^{(1)}}{2p-1}\right)^\alpha + \sum_{k=q}^{q+p-2} 2(q+p-k-1) \left(\frac{K_k^{(1)}}{2(q+p-k-1)}\right)^\alpha ,$$

and, since $K_k^{(1)} = K$ and $\sum_{k=1}^{p-1} k^{1-\alpha} \geq \int_1^p dx/x^{1-\alpha}$,

$$\mathcal{P}_{\max} \geq K^\alpha \left(2 \times \frac{1}{2^{\alpha-1}} \frac{1}{2-\alpha} \left(1 - p^{2-\alpha}\right) + \frac{q-p}{(2p-1)^{\alpha-1}}\right),$$

and hence $\mathcal{P}_{\max} = O(K^\alpha)$, since $\alpha > 2$ and $q = O(p)$.

Finally, since $\mathcal{P}_{\text{XY}} = O(p \times K^\alpha)$, we conclude that the worst ratio $\mathcal{P}_{\text{XY}}/\mathcal{P}_{\max}$ is at most in $O(p)$, hence providing us an upper bound on this ratio.

We now exhibit an instance of the problem on a $p \times q$ CMP, such that $q = O(p)$ and $q \geq p$, and a max-MP routing such that the ratio (in $O(p)$) is realized, when all communications go from the same

source core $\mathcal{C}_{1,1}$ to the same destination core $\mathcal{C}_{p,q}$. Let $p = 2 \times p'$, and K be the total size of the communications to route. The power consumed with an XY routing is $\mathcal{P}_{\text{XY}} = (p+q) \times K^\alpha$.

Now we consider the max-MP routing pattern based on Figure 5.4. Until semi-diagonal $D^{(1)}_{2p'}$, communications are split according to the figure. Then the communications that arrive (there are p' of them) at $D^{(1)}_{2p'}$ are forwarded horizontally. When they reach $D^{(1)}_{q}$, communications are aggregated according to the symmetrical pattern of the figure.

We first compute $\mathcal{P}^{(1)}_{\max}$, the dissipated power at both ends, where the communications are not forwarded horizontally. We deal with the cores in diagonal. On semi-diagonal $D^{(1)}_{2k}$, for $j \in \{1, \ldots, k\}$, the core $\mathcal{C}_{j,2k+1-j}$ on line j is sending $r_{k,j}$ communications to its right core, and $d_{k,j}$ to its down core. Between $D^{(1)}_{2k}$ and $D^{(1)}_{2(k+1)}$, for $j \in \{1, \ldots, k+1\}$, the core $\mathcal{C}_{j,2k+2-j}$ on line j is sending h_{k+1} communications to its right core.

We set:

– for $k \in \{1, \ldots, p'\}$, $h_k = \dfrac{K}{k}$;

– for $k \in \{1, \ldots, p'-1\}$ and $j \in \{1, \ldots, k\}$,

$$r_{k,j} = \frac{k+1-j}{k(k+1)}K \quad \text{and} \quad d_{k,j} = \frac{j}{k(k+1)}K \ .$$

We show that the splits and merges of communications are valid:

– for $k \in \{1, \ldots, p'-1\}$ and $j \in \{2, \ldots, k\}$,

$$r_{k,j} + d_{k,j-1} = \frac{k}{k(k+1)}K = h_{k+1} \ ;$$

– for $k \in \{1, \ldots, p'-1\}$, $r_{k,1} = h_{k+1}$ and $d_{k,k} = h_{k+1}$;

– for $k \in \{1, \ldots, p'-1\}$ and $j \in \{1, \ldots, k\}$,

$$r_{k,j} + d_{k,j} = \frac{k+1}{k(k+1)}K = h_k \ .$$

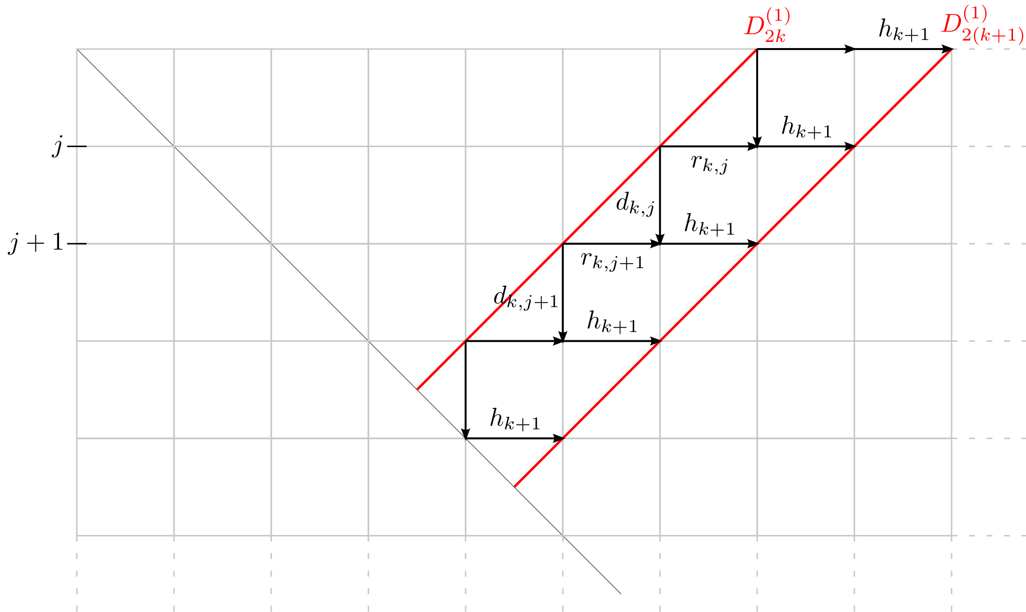


Figure 5.4: Routing pattern.

What is the dissipated power with this max-MP routing? The power consumption $\mathcal{P}^{(1)}_{\max}$ is twice the power consumed until diagonal $D^{(1)}_{2p'}$ (we define symmetrical routes for the other half of the routing). Therefore, we have:

$$\begin{aligned}\frac{1}{2}\mathcal{P}^{(1)}_{\max} &= \sum_{k=1}^{p'} k\,(h_k)^\alpha + \sum_{k=1}^{p'-1}\sum_{j=1}^{\alpha} ((d_{k,j})^\alpha + (r_{k,j})^\alpha) \\ &\le \sum_{k=1}^{p'} k\,(h_k)^\alpha + \sum_{k=1}^{p'-1}\sum_{j=1}^{k} (d_{k,j} + r_{k,j})^\alpha \ .\end{aligned}$$

Also, we know that for $k \in \{1,\dots,p'-1\}$ and $j \in \{1,\dots,k\}$, $d_{k,j} + r_{k,j} = h_k$. Therefore,

$$\begin{aligned}\frac{1}{2}\mathcal{P}^{(1)}_{\max} &\le \sum_{k=1}^{p'} k\,(h_k)^\alpha + \sum_{k=1}^{p'-1} k\,(h_k)^\alpha \le 2K^\alpha \sum_{k=1}^{p'} \frac{1}{k^{\alpha-1}} \\ &\le 2K^\alpha \left(1 + \frac{1}{\alpha-2}(1-(p')^{2-\alpha})\right) \ .\end{aligned}$$

Now, the power dissipated in the horizontal links in the middle of the CMP is:

$$\mathcal{P}^{(2)}_{\max} = (q-p)p' \times (K/p')^\alpha \le K^\alpha \times q(p')^{1-\alpha}.$$

There are indeed p' communications of size K/p', each of length $(q-p)$. Altogether,

$$\begin{aligned}\mathcal{P}_{\max} &= \mathcal{P}^{(1)}_{\max} + \mathcal{P}^{(2)}_{\max} \\ &\le K^\alpha \left(\frac{4(\alpha-1)}{\alpha-2} - \frac{4(p')^{2-\alpha}}{\alpha-2} + q(p')^{1-\alpha}\right).\end{aligned}$$

Since $q = O(p')$ and $\alpha > 2$, we have $4(\alpha-1)/(\alpha-2) - 4(p')^{2-\alpha}/(\alpha-2) + q(p')^{1-\alpha} = O(1)$, and since $\mathcal{P}_{\mathrm{XY}} = (p+q) \times K^\alpha$ and $q = O(p)$, the ratio $\mathcal{P}_{\mathrm{XY}}/\mathcal{P}_{\max}$ is in $O(p)$, which concludes the proof. ■

This shows that even with an exponential number of paths, using multi-path routing on a square CMP, in which all communications have the same source core and the same destination core, leads to a power improvement factor of up to $O(p)$, compared to an XY routing. Moreover, this factor can be reached with a max-MP routing. We did not succeed to derive this factor with a single-path routing (1-MP), and this is left as an open problem.

In the next paragraph, we investigate whether this factor can be improved when communications must be routed from/to different core pairs.

Multiple sources and multiple destinations. We now consider that several communications with different sources and destinations must be routed on the CMP. The upper bound on the improvement factor when using (multiple) Manhattan paths then becomes $O(p^{\alpha-1})$, and this ratio is reached even for a 1-MP single-path routing.

Theorem 5.2. *Given a $p \times q$ CMP with $q \ge p$, $q = O(p)$, and a set of communications, the minimum upper bound for the ratio of the power consumed by an XY routing ($\mathcal{P}_{\mathrm{XY}}$) over the power consumed by a max-*MP *routing ($\mathcal{P}_{\max}$) is in $O(p^{\alpha-1})$.*

Proof. Similarly to the proof of Theorem 5.1, we first show that an upper bound of $\mathcal{P}_{\mathrm{XY}}/\mathcal{P}_{\max}$ is in $O(p^{\alpha-1})$. The tightness result is given in Lemma 5.2, for a 1-MP routing.

We start by providing a lower bound of $\mathcal{P}_{\max}$, following the same line of reasoning as in the proof of Theorem 5.1. This time, we have to consider diagonals going into each of the four possible directions: for each $k \in \{1, \ldots, q+p-2\}$ and for each $d \in \{1, \ldots, 4\}$, $K_k^{(d)}$ is the sum of the δ_i such that $d_i = d$, $ksrc(i) \leq k$ and $ksnk(i) > k$.

For a given $K_k^{(d)}$, the *ideal* way to map those communications (with as many paths as desired) is to distribute them equally among all the communication links from $D_k^{(d)}$ to $D_{k+1}^{(d)}$, hence providing us with a lower bound on $\mathcal{P}_{\max}$. Thus, if all communications go in direction d, we have:

$$\begin{aligned}\mathcal{P}_{\max}^{(d)} \geq & \sum_{k=1}^{p-1} 2k \left(\frac{K_k^{(d)}}{2k}\right)^{\alpha} + \sum_{k=p}^{q-1} (2p-1)\left(\frac{K_k^{(d)}}{2p-1}\right)^{\alpha} \\ & + \sum_{k=q}^{q+p-2} 2(q+p-k-1)\left(\frac{K_k^{(d)}}{2(q+p-k-1)}\right)^{\alpha} \\ \geq & \frac{4}{(2p)^{\alpha-1}} \sum_{k=1}^{q+p-2} \left(K_k^{(d)}\right)^{\alpha}.\end{aligned}$$

Note that for a given communication link that is between two successive diagonals in a direction, there exists another direction such that this link is between two successive diagonals in this direction. For instance $\mathcal{L}_{(1,1)\to(1,2)}$ goes from $D_1^{(1)}$ to $D_2^{(1)}$ but also from $D_p^{(4)}$ to $D_{p+1}^{(4)}$.

However, because of the convexity of the power function, the power dissipated by a routing is less than the power dissipated if the communications in each direction would not interfere:

$$\mathcal{P}_{\max} \geq \sum_{d=1}^{4} \mathcal{P}_{\max} s^{(d)} = \frac{4}{(2p)^{\alpha-1}} \sum_{d=1}^{4} \sum_{k=1}^{q+p-2} \left(K_k^{(d)}\right)^{\alpha}.$$

There remains to find an upper bound on $\mathcal{P}_{\mathrm{XY}}$, which is more difficult to achieve than in the single source/destination case. First, for a given sum of communications $K_k^{(d)}$ and a given occupation of the links from $D_k^{(d)}$ to $D_{k+1}^{(d)}$, note that the worst case would be to map the whole $K_k^{(d)}$ onto the maximum occupied link, because of the convexity of the power function. Let us consider now the direction 1. We relax the problem by saying that the set of communication links from $D_k^{(1)}$ to $D_{k+1}^{(1)}$ has a non empty intersection with any set of links from $D_{k'}^{(2)}$ to $D_{k'+1}^{(2)}$, $k' \in \{1, \ldots, q+p-2\}$, and with any set of links from $D_{k''}^{(4)}$ to $D_{k''+1}^{(4)}$, $k'' \in \{1, \ldots, q+p-2\}$. We keep on relaxing by placing the $K_k^{(1)}$ both on a link of the first set and on a link of the second set.

Then, for $d = 2$ and $d = 4$, $\sigma_{1,d}$ is the permutation of $\{1, \ldots, q+p-2\}$ such that

$$\sum_{k=1}^{p+q-2} \left(K_k^{(1)} + K_{\sigma_{1,j}(k)}^{(d)}\right)^{\alpha}$$

is maximum. We map $K_k^{(1)}$ and $K_{\sigma_{1,j}(k)}^{(d)}$ onto the same link, thus $K_{\sigma_{1,j}(k)}^{(d)}$ cannot interfere anymore with another $K_{k'}^{(1)}$, hence the permutation.

We define $\sigma_{3,2}$ and $\sigma_{3,4}$ in the same way and obtain that:

$$\mathcal{P}_{\text{XY}} \leq \sum_{k=1}^{p+q-2} \left(K_k^{(1)} + K_{\sigma_{1,2}(k)}^{(2)}\right)^\alpha + \left(K_k^{(1)} + K_{\sigma_{1,4}(k)}^{(4)}\right)^\alpha + \left(K_k^{(3)} + K_{\sigma_{3,2}(k)}^{(2)}\right)^\alpha + \left(K_k^{(3)} + K_{\sigma_{3,4}(k)}^{(4)}\right)^\alpha.$$

Indeed, we account for all communications, in any direction. Since for all (a, b), $(a + b)^\alpha \leq (2a)^\alpha + (2b)^\alpha$, we deduce that

$$\mathcal{P}_{\text{XY}} \leq 2 \times 2^\alpha \sum_{k=1}^{p+q-2} \sum_{d=1}^{4} \left(K_k^{(d)}\right)^\alpha,$$

and hence $\mathcal{P}_{\text{XY}}$ is in $O(1)$.

Finally we conclude that the ratio $\mathcal{P}_{\text{XY}}/\mathcal{P}_{\max}$ is at most in $O(p^{\alpha-1})$. We prove that this ratio can indeed be achieved in Lemma 5.2. ■

Lemma 5.2. *The ratio in $O(p^{\alpha-1})$ of Theorem 5.2 can be achieved with a* 1-MP *routing on a square CMP.*

Proof. We consider a $p \times p$ CMP, where $p = p' + 1$, and a set of p' communications $\gamma_1, \dots, \gamma_{p'}$, where for all $i \in \{1, \dots, p'\}$, $\gamma_i = (\mathcal{C}_{1,i}, \mathcal{C}_{i,p'+1}, 1)$.

The XY routing depicted in Figure 5.5(b) has a power consumption of $\mathcal{P}_{\text{XY}} = 2\sum_{i=1}^{p'} i^\alpha$. We have:

$$(p')^{\alpha+1} \leq \frac{\mathcal{P}_{\text{XY}}}{2(\alpha+1)} \leq (p'+1)^{\alpha+1} - 1,$$

hence $\mathcal{P}_{\text{XY}}$ is in $O((p')^{\alpha+1})$.

The 1-MP routing depicted in Figure 5.5(a) is a YX routing, and its power consumption is:

$$\mathcal{P}_{1-\text{MP}} = \sum_{i=1}^{p'} 2i \times 1^\alpha = p'(p'+1).$$

We conclude that in this example the ratio $\mathcal{P}_{\text{XY}}/\mathcal{P}_{1-\text{MP}}$ is in $O(p^{\alpha-1})$, hence matching the upper bound. ■

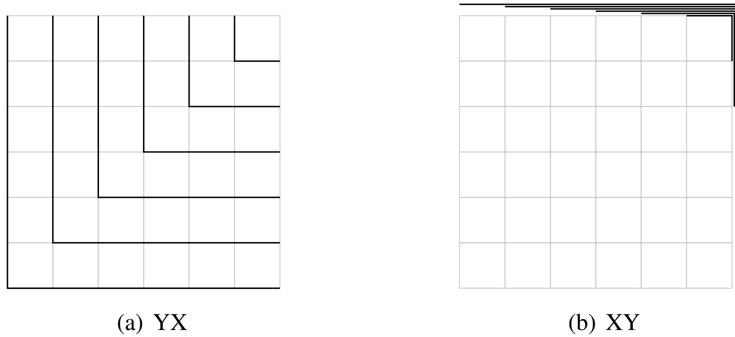


Figure 5.5: Proof of Lemma 5.2.

5.4.2 NP-completeness

Theorem 5.3. *Finding a s-MP routing that minimizes the total power consumption while ensuring that link bandwidths are not exceeded is a NP-complete problem.*

Proof. Consider the associated decision problem: given a power threshold P, is there a s-MP routing that does not exceed any link bandwidth, and such that the total power consumption is not greater than P? The problem is obviously in NP: given a routing, it is easy to check in polynomial time that it is a s-MP routing (each communication is split in at most s communications), that the bandwidth on each link is not exceeded, and that the total power consumption is not greater than P.

In fact, even without any power consideration, we prove that the problem of matching the bandwidth constraints is NP-complete. The associated decision problem is as follows: is there a s-MP routing that does not exceed any link bandwidth?

To establish the completeness, we use a reduction from 2-PARTITION. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: we are given n strictly positive integers $a_1, a_2, \dots, a_n$, does there exist a subset I of $\{1, \dots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Let $S = \sum_{i=1}^{n} a_i$.

We build an instance $\mathcal{I}_2$ of our problem. The CMP is of size $p \times q$, with $p = 2$ and $q = (s-1)n + 2$, and the maximum bandwidth of communication links is $BW = S/2 + (s-1)n$. We have $n_c = n + q$ communications $(\gamma_1, \gamma_2, \dots, \gamma_{n_c})$ to route. The first n communications are traversing the CMP: γ_1 goes from $\mathcal{C}_{1,1}$ to $\mathcal{C}_{p,q}$; γ_2 starts from $\mathcal{C}_{1,s}$, and so on: for each $i \in \{1, \dots, n\}$, $\gamma_i = (\mathcal{C}_{1,(i-1)(s-1)+1},\ \mathcal{C}_{p,q},\ a_i + s - 1)$. The last q communications are one-hop vertical communications: for each $i' \in \{1, \dots, q-2\}$, $\gamma_{n+i'} = (\mathcal{C}_{1,i'}, \mathcal{C}_{2,i'}, BW - 1)$; $\gamma_{n_c - 1} = (\mathcal{C}_{1,q-1}, \mathcal{C}_{2,q-1}, BW - \frac{S}{2})$, and $\gamma_{n_c} = (\mathcal{C}_{1,q}, \mathcal{C}_{2,q}, BW - \frac{S}{2})$.

Note that since the routing is using only shortest paths, we do not have any choice for the routing of communications $\gamma_{n+1}, \dots, \gamma_{n_c}$: each communication must follow the vertical link, as shown in Figure 5.6.

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_2$ has a solution if and only if $\mathcal{I}_1$ does. Suppose first that $\mathcal{I}_1$ has a solution and let I be a subset of $\{1, \dots, n\}$ such that $\sum_{i \in I} a_i = S/2$. For each $i \in \{1, \dots, n\}$, we split the communication γ_i into $\gamma_{i,1}, \dots, \gamma_{i,s}$ such that $\delta_{i,s} = a_i$ and for all $k \in \{1, \dots, s-1\}$, $\delta_{i,k} = 1$. To define completely a path, we just have to decide for the vertical link that is used. For each $i \in \{1, \dots, n\}$ and each $k \in \{1, \dots, s-1\}$, $\gamma_{i,k}$ uses $\mathcal{L}_{(1,(i-1)(s-1)+k) \to (2,(i-1)(s-1)+k)}$. For each $i \in I$, $\gamma_{i,s}$ uses $\mathcal{L}_{(1,q-1) \to (2,q-1)}$ and for each $i \in \{1, \dots, n\} \setminus I$, $\gamma_{i,s}$ uses $\mathcal{L}_{(1,q) \to (2,q)}$. No link bandwidth is exceeded and we obtain a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. All source cores are on line 1, all destination cores are on line 2, and the sum of all communications is equal to the total available bandwidth of the vertical links. Therefore, each vertical link must be fully utilized, up to the maximum bandwidth BW. Since communication

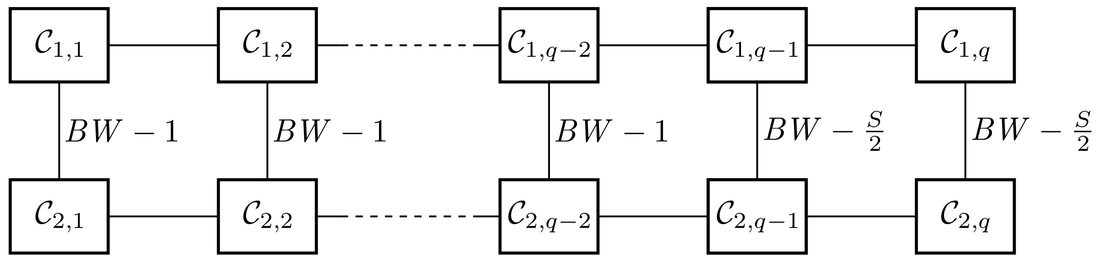


Figure 5.6: NP-completeness proof.

γ_1 is the only one that can use links $\mathcal{L}_{(1,1)\to(2,1)}$ to $\mathcal{L}_{(1,s-1)\to(2,s-1)}$, it must send a communication with $\delta_{1,k} = 1$ on each of these links, for $1 \leq k \leq s-1$. After that, this communication cannot be split anymore because the routing must use at most s paths. Because the available bandwidth of the vertical links until the last two ones is $BW-1$, the a_1 remaining bytes must wait until $\mathcal{C}_{1,q-1}$ or $\mathcal{C}_{1,q}$ to go down. We can reiterate this reasoning on the next communications $\gamma_2, \dots, \gamma_n$. Finally the 2-partition comes from the fact that at most $S/2$ bytes can go down through $\mathcal{L}_{(1,q-1)\to(2,q-1)}$ and the vertical links have to be saturated. This concludes the proof. ■

5.5 Heuristics

We present in this section several heuristics to solve the 1-MP problem. Note that we restrict ourselves to single-path routing heuristics because of the overhead incurred by routing a given communication across several paths; with the packets following different paths, reconstructing the message becomes a time-consuming task and may well involve complicated buffering policies. Instead, we envision a table-driven scheduling algorithm, which the system can safely call each time there is a new set of applications to be routed along the CMP. Furthermore, thanks to the theoretical results of Section 5.4, we hope significant gains over XY routing when using single-path routing, as is shown in Lemma 5.2.

In all the heuristics, when we deal with the communications greedily, these are sorted by decreasing number of bytes per second δ_i, which we call *weight* in the following. We have considered variants of the heuristics, where communications are sorted according to another criterion (as for instance their length, or the ratio of their weight over their length). It turns out that decreasing weights gives the best results, hence we report only this variant. The source code for all heuristics and simulations is available at [101].

5.5.1 Simple greedy (SG)

We route communications one by one, and for each communication, we build the path from the source core to the destination core hop by hop, the next used link being the least loaded link among the one or two possible next links. If there is a tie, we choose the link that gets closer to the diagonal, from the source core to the sink core.

5.5.2 Improved greedy (IG)

We pre-route the communications as if all possible links between two diagonals could be used and if we could share each communication among all those links, similarly to Figure 5.3. As mentioned in Section 5.4.1, such a pre-routing cannot be achieved, and we merely use it as a virtual initial distribution. We sort the communications by decreasing weights, and deal with the communications greedily.

When processing a communication γ_i, we first remove all its contributions to the loads of the links (remove its pre-routing) and then find a unique route for this communication (with the pre-routing loads of the yet un-processed communications still on the links). Starting from the source core, we choose at each step the next link that will be used in the following way (there are at most two possible links). Recall that d_i is the direction of γ_i, and let k_0 be such that the current core $\mathcal{C}_{u,v}$ belongs to $D_{k_0}^{(d_i)}$. If $u = usnk(i)$ (resp. $v = vsnk(i)$), we have no choice, the next link is horizontal (resp. vertical). Otherwise, we choose the one of the two links between diagonals $D_{k_0}^{(d_i)}$ and $D_{k_0+1}^{(d_i)}$ that could lead to the lowest power consumption. For each of the two possible links, we compute a lower bound on the power consumption to reach the sink core after the chosen link: for each $k \in \{k_0+1, \dots, usnk(i)+vsnk(i)-1\}$, we keep the least loaded possible link between $D_k^{(d_i)}$ and $D_{k+1}^{(d_i)}$, and we compute the power consumption

if we add communication γ_i. The lower bound is obtained by summing all these power consumptions, together with the power consumption of the link chosen between $D_{k_0}^{(d_i)}$ and $D_{k_0+1}^{(d_i)}$. Finally, we choose the link with the smallest lower bound, and we iterate until the destination core is reached.

5.5.3 Two-bend (TB)

We authorize at most two bends for the routing of a given communication. Once again, we sort the communications by decreasing weights. For each communication γ_i, we try all possible routings (there are at most $|usrc(i) - usnk(i)| + |vsrc(i) - vsnk(i)|$ different two-bend routings), and we keep the best one (in terms of power consumption).

5.5.4 XY improver (XYI)

The idea is to start with an XY-routing and to try to decrease the load of the most loaded links. We first route the communications using XY-routing, and we build a list of links, containing all the links, from the most loaded one to the least loaded one. We take the first link in the list. For each communication going through this link, we try to move it, so that it avoids this highly loaded link. More precisely, if the link is vertical, we use instead the horizontal link going to the same core, from the core that is the closest to the source core of the communication. If the link is horizontal, we instead use the vertical link going from the same core, and going to the core that is closest to the sink core of the communication. If the communication cannot be moved without violating the Manhattan path constraint, it is removed from the list of the communications going through this link.

For each communication, we compute the power consumption with the modified routes. If none of the modifications lead to a lower power consumption (or simply if no modification is available), we remove the link from the list, and iterate with the next link in the list. If at least one modification leads to a power improvement, we keep the new routing that consumes the lowest power, update the load of the links, and we sort again the list of links by decreasing load. We then iterate. Note that there are at most $p \times q$ modifications per communication.

5.5.5 Path remover (PR)

Similarly to heuristic **IG**, we first assume that each communication is (virtually) pre-routed with all paths from its source node to its destination node, as in Figure 5.3. Then, we iteratively remove links for the communications, until there remains only one path for each of them. While there remains a communication with two or more paths, we consider the most loaded link, and the largest communication that uses this link. We remove this link from the list of links used by this communication, unless this removal would break its last remaining path for this communication. Otherwise, we consider removing the second communication, and so on.

After removing a link for a communication γ_i, we need some path cleaning operation. We update the array of possible links for γ_i (initially, it contains all Manhattan paths), in such a way that it is easy to check, when considering a subsequent deletion, if there remains a path for γ_i. For example, assume that $d_i = 1$. If we delete $\mathcal{L}_{(u,v)\to(u,v+1)}$, and if the link $\mathcal{L}_{(u,v)\to(u+1,v)}$ has already been removed, we delete as well the links $\mathcal{L}_{(u-1,v)\to(u,v)}$ and $\mathcal{L}_{(u,v-1)\to(u,v)}$. Also, if we delete $\mathcal{L}_{(usrc(i),v)\to(usrc(i),v+1)}$, then all the links $\mathcal{L}_{(usrc(i),v')\to(usrc(i),v'+1)}$ for all $v' \in \{v, \dots, vsnk(i)-1\}$, and $\mathcal{L}_{(usrc(i),v'')\to(usrc(i)+1,v'')}$ for all $v'' \in \{v, \dots, vsnk(i)\}$, can be deleted. Finally, we can remove a link between diagonals $D_k^{(d)}$ and $D_{k+1}^{(d)}$ only if there are at least two valid links between those two diagonals. Please refer to [101] for further details on the implementation.

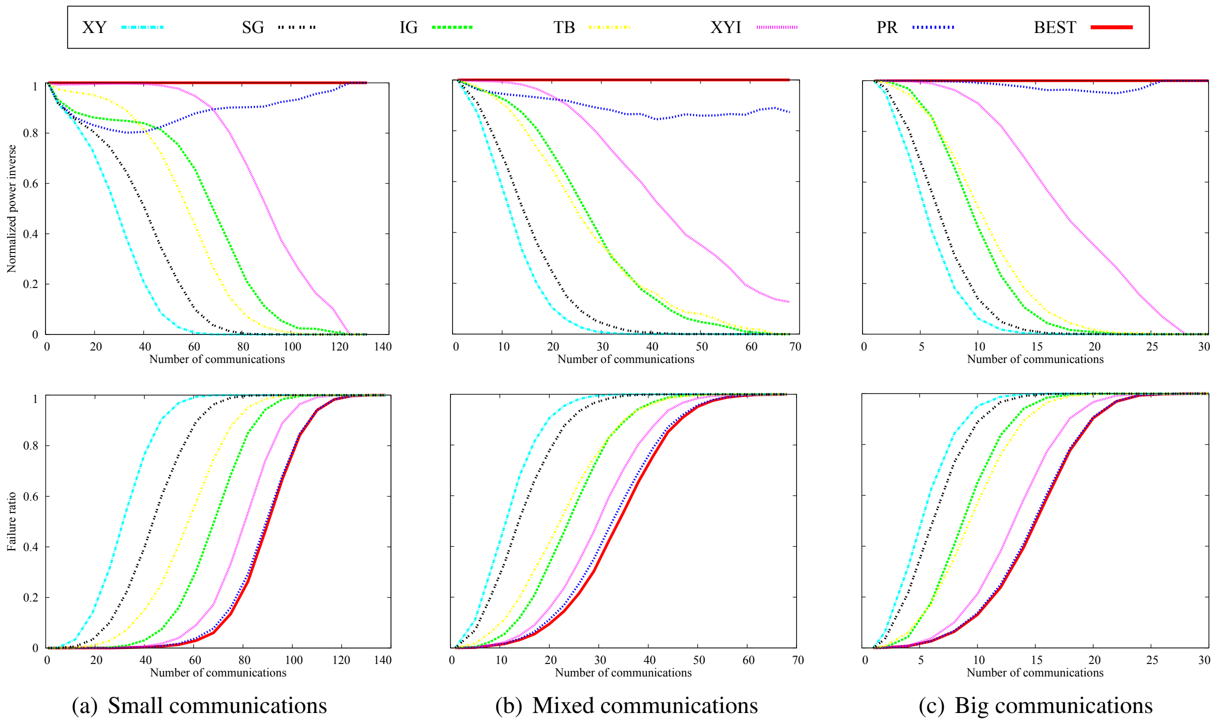
5.6 Simulations

As mentioned earlier, the source code for the simulations is available at [101]. The CMP is of size 8×8. Given that implementing continuous frequencies is not practical, we use the characteristics of the links described in [69]. The given discrete values for the frequencies fit our model with $P_{\text{leak}}^{(\text{comm})} = 16.9\,\text{mW}$, $P_0 = 5.41$ and $\alpha = 2.95$. We have then three possible frequencies: $1\ \text{Gb/s}$, $2.5\ \text{Gb/s}$ and $3.5\ \text{Gb/s}$. Note that the heuristics presented in the previous section work with both continuous frequencies and discrete frequencies; in this latter case (which is the case of these simulations), each time that we compute the power consumption, we pick the first frequency in the set of possible frequencies higher than the required continuous frequency. We use random source and sink nodes for the communications.

In addition to the heuristics described in Section 5.5 (**SG**, **IG**, **TB**, **XYI**, **PR**), we run the **XY** heuristic, and we define the **BEST** heuristic as the best heuristic among all six ones on the given problem instance. Each point of the graph is obtained by averaging on 50000 sets of communications. For each simulation, we plot the inverse of the power of each heuristic (which we set to 0 if the heuristic fails), that we normalized by the inverse of the power of **BEST**, and the ratio of failures (instances where the heuristic does not find a solution).

5.6.1 Sensitivity to the number of communications

We first assess the impact of the number of communications, for both small, mixed and big communications. Results are reported in Figure 5.7.



(a) Small communications (b) Mixed communications (c) Big communications

Figure 5.7: Sensitivity to the number of communications.

Small communications

We draw the weight of each communication uniformly between $100\,\mathrm{Mb/s}$ and $1500\,\mathrm{Mb/s}$. Concerning the capacity of the heuristics to find a solution, the failure ratio defines a clear hierarchy among the heuristics. From the worst one to the best one, we have **XY**, **SG**, **TB**, **IG**, **XYI** and finally **PR**. **XY** begins to fail with less than 10 communications. With 80 communications, **XY** and **SG** fail almost all the time, while **PR** succeeds four times out of five, **XYI** half the time, **IG** every fifth time and **TB** every tenth time. **PR** succeeds almost every time when at least one heuristic succeeds.

The power inverse keeps this hierarchy, except that **PR** is not the best heuristic when the constraints are low, because it does not care about static power. **PR** stays at 80% of **BEST** for any number of communications, but **XYI** is the best heuristic when there are less than 70 communications, and then its performance drops.

Mixed communications

We draw the weight of each communication uniformly between $100\,\mathrm{Mb/s}$ and $2500\,\mathrm{Mb/s}$. With these parameters, we reach more or less the same conclusions, except that **TB** and **IG** now have almost the same results.

Big communications

We draw the weight of each communication uniformly between $2500\,\mathrm{Mb/s}$ and $3500\,\mathrm{Mb/s}$. With such large communications, **PR** is still the best heuristic, and it is closer to **BEST** than previously: it is always within 95% of **BEST**.

5.6.2 Sensitivity to the size of communications

Here we study the behavior of the heuristics, when the size of communications gets larger, for three different sizes of the communication set. Results are reported in Figure 5.8.

Few communications

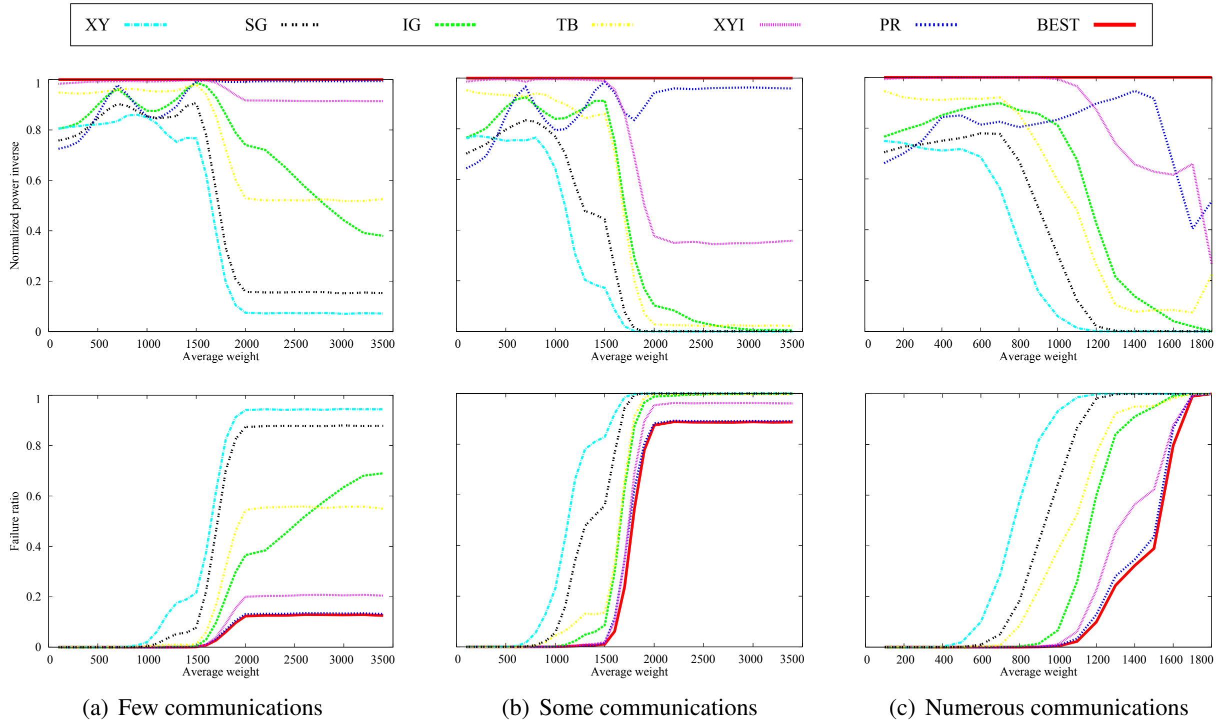
In this experiment, we draw 10 communications. **XYI** is clearly the best heuristic if the average weight is less than $1600\,\mathrm{Mb/s}$, otherwise **PR** is the best: in their best range, their inverse power always is up to 98% of **BEST**. One can remark that the performance of all heuristics is suddenly decreasing around $1750\,\mathrm{Mb/s}$. This comes from the fact that as soon as the weight of every communication reaches $1751\,\mathrm{Mb/s}$, then two communications cannot share the same link any more.

Some communications

We now draw 20 communications. Even though **XYI** is always at 99% of **BEST** when the average weight is less than $1750\,\mathrm{Mb/s}$, it falls at only 35% of **BEST** for weights larger than $2000\,\mathrm{Mb/s}$. Conversely **PR** is not affected.

Numerous communications

Finally we draw 40 communications. Here **XYI** is at 90% of **BEST** until $1100\,\mathrm{Mb/s}$, and then falls down. **PR** is always at 60% of **BEST**.



(a) Few communications (b) Some communications (c) Numerous communications

Figure 5.8: Sensitivity to the size of communications.

5.6.3 Sensitivity to the average length of communications

Finally, we study the influence of the length of the communications, i.e., the Manhattan distance between the source core and the destination core, on the performance of the various heuristics. In both previous simulation sets, we have drawn the source core and the sink core randomly, regardless of the length of the communication. Now we draw only communications whose length is around the target average length. Results are reported in Figure 5.9.

Numerous small communications

We draw 100 communications, whose weight is between $200\,\mathrm{Mb/s}$ and $800\,\mathrm{Mb/s}$. We see that **XYI** is the best heuristic until the average length is 10, and stays at least within 90% of **BEST**. Moreover, **PR** is around 80% of **BEST** before a length of 10 and then becomes the best heuristic.

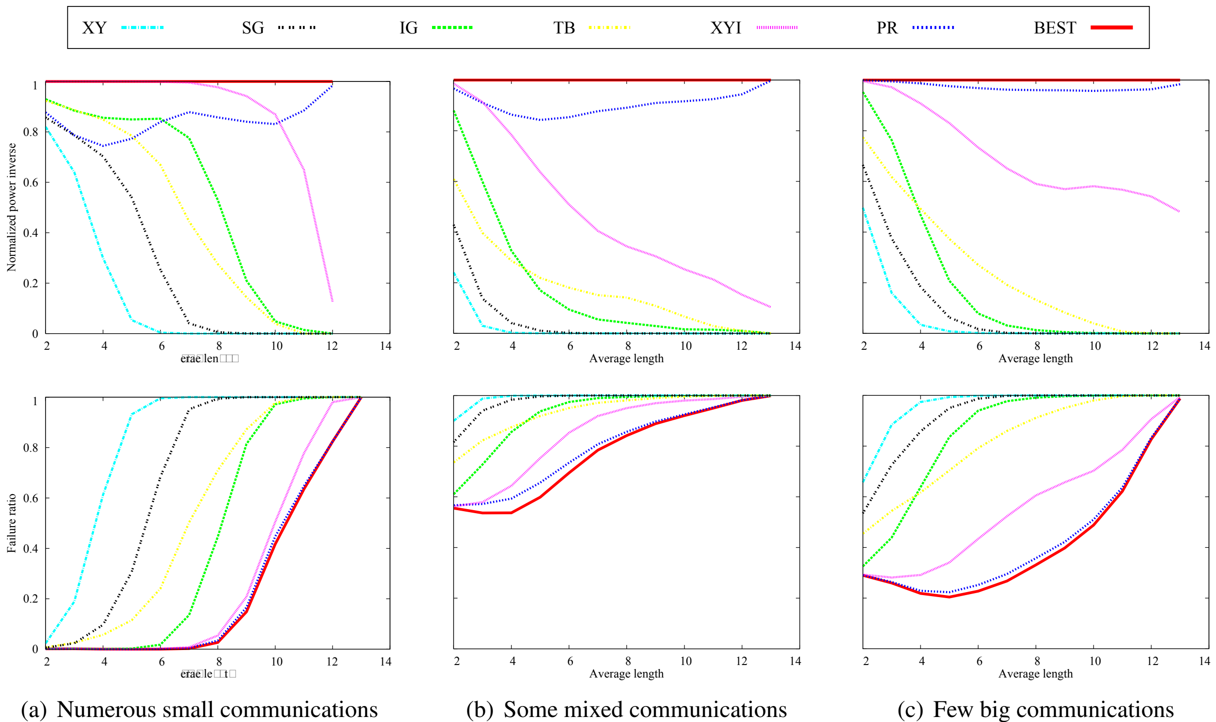
Some mid-weighted communications

We draw 25 communications, whose weight is between $100\,\mathrm{Mb/s}$ and $3500\,\mathrm{Mb/s}$. Except for a length of 2, **PR** is the best heuristic, and stays at least within 85% of **BEST**. We observe that **XYI** is the second best heuristic, decreasing regularly from 95% to 10%.

Few big communications

We draw 12 communications, whose weight is between 2700 Mb/s and 3300 Mb/s. For any length, **PR** is the best heuristic, within about 90% of **BEST**. Compared to **BEST**, **XYI** decreases from 95% to 40%. **IG** is slightly better than **TB** for communications of length less than 5, and after that, **TB** is better than **IG**.

The number of failures of **BEST** decreases from communications of length 2 to communications of length 5: this is because short communications are more likely to occur on X-axis or Y-axis; in this case, if two communications are on the same axis, we do not have any choice to separate these communications.



(a) Numerous small communications (b) Some mixed communications (c) Few big communications

Figure 5.9: Sensitivity to the length of communications.

5.6.4 Summary of simulations

Altogether, **XYI** and **PR** are the best two heuristics: **XYI** is better than **PR** when the problem is not severely constrained, but **PR** is more and more competitive, compared to the other heuristics, when the problem becomes constrained. This last observation holds true for any constraint type, be it a high number of communications, or heavily-weighted communications. **TB** is slightly better than **IG** in almost all situations, and these heuristics return a solution in fewer cases; in addition, whenever they succeed, their solution is worse than those of **XYI** and **PR**. Finally, **SG** improves the solution given by **XY**, but this solution is far from **BEST**.

On average, over all problem instances, **XY** succeeds only 15% of the times, while **XYI** and **PR** succeeds respectively 46% and 50% of the times. This last value confirms that **PR** is the best heuristic to find a valid solution, because **BEST** succeeds 51% of the times. A first conclusion is that Manhattan routing finds three times more solutions than XY routing, which is a very significant result.

Concerning the absolute inverse of power consumption, its average value is 2.44 (resp. 2.57) times higher in **XYI** (resp. **PR**) than in **XY**, and even 2.95 times higher in **BEST**. Moreover, this dramatic gain of energy is achieved within quite a reasonable time: in average, the solution is obtained in $24\,\mathrm{ms}$ for **XYI**, and in $38\,\mathrm{ms}$ for **PR**.

We conclude this section with an interesting statistical value: averaging over all the experiments, static power accounts for $1/7$-th of the total power (and dynamic power accounts for the remaining $6/7$-th fraction). These fractions obviously depend upon (i) the absolute values of the parameters, and (ii) the total communication volume. For instance a lower value of the ratio $P_{\mathrm{leak}}^{(\mathrm{comm})}/P_0$ would favor **PR** over other heuristics.

5.7 Conclusion

In this chapter, we have investigated the problem of routing communications in chip multiprocessors. While the most natural and widely used algorithm to handle communications is XY routing, we have shown that the consumed power can be dramatically reduced when using Manhattan routing instead of XY routing, and this with both a theoretical and a practical perspective.

On the theoretical side, we establish the NP-completeness of the problem of finding a Manhattan routing that minimizes the dissipated power, and we exhibit the minimum upper bound of the ratio of the power consumed by an XY routing over the power consumed by a Manhattan routing. We consider either that multiple paths may be used to route a single communication, or that a unique Manhattan route must be chosen (single-path). When several concurrent communications should be routed, it turns out that the worst case ratio of power consumption can be achieved even when restricting to single-path Manhattan routing.

On the practical side, we design several single-path polynomial time heuristics, and we compare them through extensive simulations. The use of a Manhattan path allows us to find valid routing solutions more than three times more often than the XY routing. Moreover, the power consumed by a Manhattan routing is always much lower than that consumed by an XY routing. Thanks to our two best heuristics, **XYI** and **PR**, power efficient solutions can be achieved in a reasonable time.

As future work, we still need to estimate how much can be gained by a single-path Manhattan routing when all communications share the same source and destination nodes. It would be very interesting to find some approximation algorithms, even though it seems quite a difficult task. Also, we would like to establish a bound on the optimal solution for single-path Manhattan routings (or even compute the optimal solution for small problem instances), so that we could give an insight on the absolute performance of our heuristics. Finally, it may be interesting to design multi-path heuristics, since these may allow for an even better load-balance of communications throughout the CMP. Of course, one would then need to account for their potential overhead at the system level.

Chapter 6

Assessment of bi-criteria heuristics for general directed acyclic graphs

6.1 Introduction

In the previous chapters, we have determined the complexity of several problems, and we have derived multiple algorithms to find approximate or optimal solutions under various (although related) energy models. In addition, we have assessed the performance of our algorithms through multiple simulations.

In this chapter, we target a more general problem than before, in that we deal with arbitrary DAGs (Directed Acyclic Graphs). We still aim at solving a makespan/energy bi-criteria problem, but we adopt a more practical approach. We have two main goals: (i) assess the performance of well-established heuristics through simulations; and (ii) assess the relevance of two different execution models.

As in previous chapters, the makespan is fixed and the objective is to minimize the energy consumption without violating the makespan constraint.

Processors are equipped with the Dynamic Voltage and Frequency Scaling (DVFS) and can run at any speed belonging to a given discrete set of speeds. We have two models to execute the tasks: VDD-HOPPING and NO-VDD-HOPPING. When VDD-HOPPING is allowed, the processor that a task is assigned can change its speed during its execution, whereas in the NO-VDD-HOPPING strategy, the speed at which a task runs must be unique.

In this chapter, we study a commonly used technique called *slack reclamation*. We are given a set of dependent tasks, whose mapping onto processors has already been decided, and we aim at changing the frequencies used for running these tasks, in order to minimize the energy consumption, without increasing the makespan nor violating a dependency constraint. The name *slack reclamation* comes from the fact that processors stretch the tasks assigned to them so as to fill in idle times, and to decrease the energy consumption. A lot of papers have studied slack reclamation, through different energy and speed models, but to the best of our knowledge, slack reclamation algorithms have not been compared extensively yet. We aim at conducting a detailed comparison: we select some well-known algorithms from the literature, we unify the models found in the papers, and we adapt the algorithms as fairly as possible, by ensuring that the choices made in the algorithms are consistent with the new common model. A set of simulations allows us to compare the performance of the algorithms, as well as the difference between the two strategies (VDD-HOPPING and NO-VDD-HOPPING).

The rest of the chapter is organized as follows: in Section 6.2 we review the work that has been done on this particular problem or on closely related ones. The framework of this study is developed in Section 6.3. Then we describe the three heuristics that we have implemented, in Section 6.4, as well as

two linear programs that give the optimal solutions. We study the performance of the four competitors through a set of simulations in Section 6.5.

6.2 Related work

We start by listing several often quoted papers, whose subject is close to the problem under study in this chapter, and explain why we did not implement the corresponding algorithms. In [131], tasks are independent, and provided with their worst-case execution times. Authors derive an algorithm that minimizes the energy consumption by exploiting both leakage energy saving and DVFS. They reclaim the static slack, as well as the dynamic slack coming from an actual execution time smaller than the worst-case execution time. In [130], independent tasks and dependent tasks without communication costs are considered, but the algorithms focus only on reclaiming the dynamic slack. The model described in [119] assumes that the dynamic part of the energy function is paid even if the considered processor is idle or communicating. Authors also propose to downgrade it in those cases, and simulations show drastic decreases in the energy consumption. In [128], the authors tackle the problem of mapping a DAG of tasks, provided with individual deadlines, onto a set of processors, that can run at continuous speeds. They design an integer program, whose constraints are linear and objective function is convex, that gives the optimal solution in polynomial time. Finally, the authors of [78] study the mapping of a DAG onto processors equipped with DVFS, but they do not try to minimize the consumed energy under a makespan constraint. They minimize simultaneously energy and makespan, and VDD-HOPPING is not allowed.

The following papers describe the heuristics that we decided to implement and submit to several performance tests. They all deal with the problem of scheduling a DAG of tasks, that has already been mapped onto a set of processors. The aim is to minimize the consumed energy under a makespan constraint, and despite a few differences, their energy models are all adaptable to a common one, which will be described in Section 6.3.4. In [14], continuous speeds are considered and the performance of the **LPHM** algorithm is assessed through a quick set of simulations, but they can easily be turned into discrete speeds. In [71], the processors are able to run at a given set of discrete speeds, and the described algorithm (**SRP**) is experimentally tested on a master-work program and a tree-based program. The algorithm **LEneS** designed in [50] is an exponential time algorithm that finds a solution under the VDD-HOPPING model. Simulations reveal the quality of the solutions and execution time of the algorithm. Authors show in [8] that the problem with VDD-HOPPING is polynomial and exhibit a linear program that returns the optimal solution **Opt** in polynomial time.

6.3 Framework

6.3.1 DAG

We consider an application in the form of a task graph. We are given a weighted graph $G = (\mathcal{V}, \mathcal{E})$ with $n = |\mathcal{V}|$. Nodes are tasks and they are denoted by $T_1, \ldots, T_n$; the weight w_i of task T_i corresponds to the required number of operations for the task execution. A communication between two tasks T_i and T_j corresponds to an edge $T_i \to T_j$ in the graph, and its communication time is given in seconds by the weight $\delta_{i \to j}$ of the edge.

6.3.2 Platform

The platform is composed of p fully interconnected processors $\mathcal{P}_1, \ldots, \mathcal{P}_p$. Communication links between processors are homogeneous; if there is a dependency $T_i \to T_j$ and if those tasks are mapped onto different processors, the communication lasts $\delta_{i \to j}$ seconds, whatever the processors; otherwise the communication does not occur, since the data is already present in the processor.

Processors are homogeneous and able to use the Dynamic Voltage and Frequency Scaling (DVFS) technique: we are given a set of n_f possible speeds $\{s_1, \ldots, s_{n_f}\}$ at which a processor can run. Those speeds are expressed in number of operations per second. The time to execute w operations at speed s_j is naturally w/s_j seconds.

6.3.3 Frequency scaling strategies

We define two frequency scaling strategies: VDD-HOPPING and NO-VDD-HOPPING. In the VDD-HOPPING strategy, a task can be split into several subtasks, and each of those subtasks can be run at its own frequency. It has been shown (e.g., in [9]) that at most two different frequencies are necessary: given a task T_i and a feasible execution time for this task, the energy consumption can be minimized by running the task at at most two different frequencies. Also, let the highest frequency of task T_i be $speed_i^{(h)}$ and the lowest one $speed_i^{(l)}$. The number of operations executed at speed $speed_i^{(h)}$ is noted $w_i^{(h)}$ while $w_i^{(l)}$ operations are done at the lowest frequency.

On the contrary, in the NO-VDD-HOPPING strategy, the speed of the processor that a task is assigned cannot be changed during this task, and this speed is denoted by $speed_i$.

6.3.4 Energy model

We come back in this chapter to the model of Chapter 1, where the static energy, i.e., the energy for a processor to be on, is neglected. The deadline is fixed, and we can assume that the energy saved if we turn off a processor when it finishes all its assigned tasks, is insignificant. Therefore we keep all processors on until the deadline, and the objective function is reduced to the dynamic power. We consider that the power dissipated by a processor, when it runs at the frequency s and when it is supplied by a voltage V, is given by: $P = V^2 \times s$. A classical approximation assumes that V is proportional to s. Also P can be computed, under a constant factor, as the cube of the frequency. We saw that w operations executed at a frequency s need w/s seconds to be done, which leads to the following expression for the energy consumed to run w operations at frequency s: $E(w, s) = w/s \times s^3 = w \times s^2$.

With the previous notations, in the NO-VDD-HOPPING strategy, the energy consumed by task T_i is $E_i = w_i \times speed_i^2$, whereas in the VDD-HOPPING strategy, this energy is given by:

$$E_i = w_i^{(h)} \times \left(speed_i^{(h)}\right)^2 + w_i^{(l)} \times \left(speed_i^{(l)}\right)^2.$$

In both cases the total energy consumed by the platform when running the entire task graph is:

$$E = \sum_{i=1}^{n} E_i.$$

6.4 Slack reclamation algorithms

6.4.1 Mapping algorithm: HEFT

The mapping phase is done thanks to a well-known algorithm: **HEFT** [114]. This is a list-scheduling algorithm that ranks tasks according to their distance to the exit node, taking communications into account. Then each task is mapped onto the processor on which it finishes first, given already taken decisions. We keep the solution of **HEFT** as a set of task sequences, where each sequence is the ordered list of tasks that are run on the same processor.

After calling **HEFT** on the initial graph, we operate some edge modifications to facilitate the work of slack reclamation algorithms. Those algorithms will not be aware of the mapping, and will only be given a modified task graph. To do that, if task T_i and task T_j are consecutive in one of the above sequences, we add an edge of communication time $\delta_{i \to j} = 0$ in the initial graph. If such an edge is already present, we just zero out the communication. We also remove useless edges: if there is an edge between two tasks that are mapped onto the same processor, but that are not consecutive, we delete this edge.

We introduce in the next subsection four algorithms, each of them being derived in two variants: one for VDD-HOPPING strategy and one for NO-VDD-HOPPING strategy. We order them according to the size of the task subsets that they consider. The first heuristic, **LPHM**, slows tasks, one after the other, looking at the children tasks. **SRP** works on paths in the graph, among which slack is distributed equally, while **LEneS** uses an estimation of the energy consumed in the entire graph to lead its choices. Finally, variants of **Opt** are linear programs that find the optimal solution in every strategy.

6.4.2 LPHM

This algorithm was published in [14]. We first compute the earliest start times of each task, and consider that those times will be the final start times. Then we iterate on tasks: we stretch each task, so that all of its direct successors can begin at their earliest start times.

Algorithm 10: **LPHM**

$computeEST()$;
forall $i \in \{1, \dots, n\}$ **do**
    $start_i = EST_i$;
forall $i \in \{1, \dots, n\}$ **do**
    $end_i = \min_{T_{i'} \in children(T_i)} (start_{i'} - \delta_{i \to i'})$;

In the initial paper, speeds are continuous; therefore the speed of each processor is chosen such that the task T_i lasts $end_i - start_i$ seconds if $w_i/s_1 \geq end_i - start_i$, and $speed_i = s_1$ otherwise. In the VDD-HOPPING model, we choose the speed and weight couples that lead to such a duration, while in the NO-VDD-HOPPING model, we set the speed at the first speed higher than the speed we would choose in the continuous model.

6.4.3 SRP

At each step of the algorithm, we decide for the speed at which will be run a task. This task is chosen as the first task of the longest path in the DAG, and is slowed down, taking a part of its slack existing on

this path. The idea here is to distribute the existing slack fairly among the tasks belonging to the longest path.

We note EST_i, LFT_i and d_i respectively the Earliest Start Time, the Latest Finish Time and the current duration of task T_i. The slack of task T_i is defined as: $slack_i = LFT_i - d_i - EST_i$. At a given step, a task is called *defined* if its speed has already been chosen during a previous step or if its slack is equal to 0. In this later case, the task has to be run at the highest frequency. The path $path_i$ of task T_i is computed as the maximum sum of duration of tasks on a path from T_i to a defined task. The main procedure of **SRP** is described in [71] and Algorithm 11 (we note $D$the deadline).

Algorithm 11: **SRP**

```
forall i ∈ {1,...,n} do
    d_i = w_i/s_{n_f}
start_1 = 0 ;
end_n = D ;
updateEST(T_1) ;
updateLFT(T_n) ;
while there is an undefined task do
    updatePaths();
    Choose the task T_i with the longest path_i ;
    stretch(T_i) ;
    updateEST(T_i) ;
    updateLFT(T_i) ;
```

A call to $updateEST(T_i)$ updates the Earliest Start Time of all T_i's successors. If T_i is stretched, i.e., we increase its duration, the EST of tasks $T_j \in children(T_i)$ may be delayed. In the same way children of each T_j might be delayed, and so on, for all successors of T_i. In turn, we update $slack_j$ for all j such that $T_j \in successors(T_i)$, and mark T_j as *defined* if its slack is equals to 0. In the same way when $updateLFT(T_i)$ is called, the Latest Finish Time and slack of all T_i's ancestors are updated. Concerning $updatePaths()$, this procedure updates the $path_i$, thanks to a run through the DAG in reverse order.

As explained previously, at each iteration, this heuristic tries to share the slack among the longest path, by consuming a part of the slack in the first task. This is done through $stretch(T_i)$, which sets the frequency $speed_i$ of task T_i. This frequency is computed in the continuous model as:

$$speed_i = \frac{path_i}{path_i + slack_i} \times s_{n_f}.$$

This formula ensures that no deadline will be missed. The constraint that must be fulfilled, so that task T_i is finished before its LFT is:

$$speed_i \geq \frac{w_i}{\frac{w_i}{s_{n_f}} + slack_i},$$

where we recall that s_{n_f} is the maximum frequency. Since $path_i \geq w_i/s_{n_f}$, all tasks will finish before their Latest Finish Time and the makespan will be matched. We choose the speed in the NO-VDD-HOPPING model and the speed and weights couples in the VDD-HOPPING model like in the **LPHM** algorithm: we take the speed couple that emulates $speed_i$ with VDD-HOPPING, and the lowest speed such that the task lasts less than $w_i/s_{n_f} + slack_i$ with NO-VDD-HOPPING.

We take a small example (see Figure 6.1) to observe the advantage of this choice for the frequency. Let G a DAG composed of $n = k + 3$ tasks, and a platform of two processors. Tasks T_2 to T_{k+1} are

arranged onto a chain mapped onto processor $\mathcal{P}_2$. The chain $T_1 \rightarrow T_{k+2} \rightarrow T_{k+3}$ is mapped onto $\mathcal{P}_1$, and we have the additional dependencies: $T_1 \rightarrow T_2$ and $T_{k+1} \rightarrow T_{k+3}$. We zero out all communications, and weights of T_1 and T_{k+3}. We force $w_{k+2} \geq \sum_{i=2}^{k+1} w_i$. When we launch Algorithm 11, we obtain that all tasks T_i, for $i \in \{2, \dots, k+1\}$ are run at speed

$$speed_i = \frac{\sum_{i'=2}^{k+1} w_{i'}}{w_{k+2}},$$

giving an optimal solution for this problem instance.

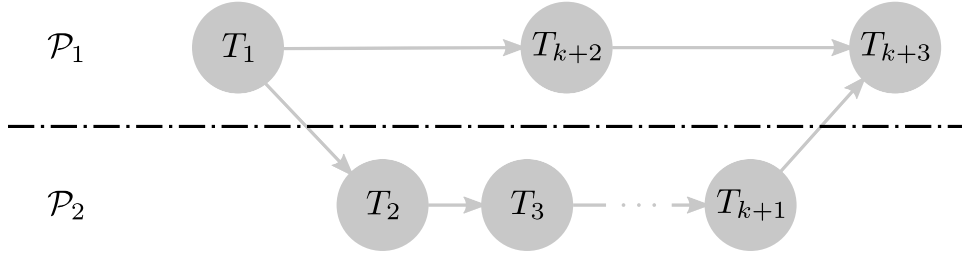


Figure 6.1: Example for **SRP**.

We can remark that at each iteration, the task T_i with the highest $path_i$ is necessarily a task whose parents are all *defined*. Let a task T_{i_1} which has a parent T_{i_2} not *defined*; then $path_{i_2} \geq path_{i_1} + w_{i_2}/s_{n_f} \geq path_{i_1}$. This observation is useful to speed up the algorithm by maintaining a set of tasks which have only *defined* parents.

6.4.4 LEneS

LEneS$_{\text{VDD}}$

This heuristic relies on the notion of *partial schedule*. In a partial schedule, each task is assigned an interval of possible durations. In addition, if we choose the duration of a given task in its interval, then there must exist, for each task, a duration in its interval, so that the execution fulfils the deadline constraint.

The initialization phase of the algorithms computes an interval for each task, based on Earliest Start Times and Latest Finish Times. Then we iteratively reduce the length of those intervals, until reaching a single possible duration for all tasks.

When we choose to prevent a task from using high frequencies, the interval of this task is impacted, as well as the interval of its successors. On the one hand, the interval of the given task will be cut from its smallest possible durations. On the other hand, its successor tasks may be constrained to begin later, hence cutting their interval from the highest possible durations. The choice to slow down a task or not is made according to a global energy estimation.

The estimation of the energy consumption of a task is related to the average energy on the interval of possible durations. Given a task T_i, let a and b be such that $w_i/s_{n_f} \leq a < b \leq w_i/s_1$. The average energy consumption of T_i on the interval $[a, b]$ is given by:

$$\overline{E_i}([a, b]) = \frac{1}{b-a} \int_a^b E_i(t)\, dt,$$

where $E_i(t)$ is the energy consumed by the task if it lasts t seconds. Then we compute the current energy estimation of task T_i by:

$$E_i^{(est)} = \overline{E_i}\left(\left[end_i - start_i\,,\,\min\left(\frac{w_i}{s_1}, LFT_i - start_i\right)\right]\right),$$

where end_i and $start_i$ are respectively the current finish and start times of tasks T_i. The global energy estimate is then $E^{(est)} = \sum_{i=1}^{n} E_i^{(est)}$.

In Algorithm 12, we iterate on the time; each step is looking at the interval of time $[t, t + timeStep]$ and tries to slow down tasks whose end time is currently in this interval. We keep in the array $nextInLine$, of size p, the first unscheduled task on each processor. $eligible$ is a list of tasks that belongs to $nextInLine$, and whose end_i is less than $t + timeStep$.

Algorithm 12: **LEneS**$_{\text{VDD}}$

```
computeEST() ;
computeLFT() ;
forall i ∈ {1, ..., n} do
    start_i = EST_i ;
    end_i = start_i + w_i/s_{n_f} ;
update(nextInLine);
E^(est) = computeEnergy() ;
t = 0 ;
while t + timeStep ≥ D do
    eligible = findEligible(nextInLine) ;
    while eligible is not empty do
        tryDelay() ;
        eligible = findEligible(nextInLine) ;
    t = t + timeStep ;
```

Algorithm 13: $tryDelay()$, version 1

```
forall i ∈ eligible do
    stretch(T_i) ;
    E_i^(ifStr) = computeEnergy();
    rollback();
i_min = argmin_i E_i^(ifStr) ;
if E_{i_min}^(ifStr) < E^(est) then
    stretch(T_{i_min}) ;
    if end_{i_min} < t + timeStep then
        schedule(T_{i_min}) ;
    E^(est) = E_{i_min}^(ifStr) ;
else
    schedule(T_{i_min}) ;
```

A call to $stretch(T_i)$ delays the end time of task T_i. More precisely, this procedure call sets $end_i = \min(t + timeStep, LFT_i)$, hence ensuring that the deadline will not be missed. This delay has an effect onto T_i's successors: the start time of a child may be shifted if T_i becomes its critical parent. The updates of successors start time is done in $computeEnergy()$, which uses those new end and start times to compute the energy estimate of all updated tasks.

In the $tryDelay()$ procedure of Algorithm 13, we try to stretch all eligible tasks, and choose the task that leads to the lowest energy estimation. If the energy is reduced when we slow down the task, we confirm the end time change, else the task is scheduled under its previous end time. One can notice that if the task $T_{i_{\min}}$ is ending at its LFT, $schedule(T_{i_{\min}})$ is called, replacing in turn $T_{i_{\min}}$ by the next task mapped onto the same processor in $nextInLine$, if any.

This implementation appears to be the closest one to the initial algorithm described in [50]. For each possible stretching, we compute the resultant energy estimation, and choose to definitively stretch (or not) the task that leads to the lowest energy. Then, if we try again to stretch the previous tasks, this can give energy estimations different from the previous iteration. But energy estimation cost is very high, also we wrote a variant of $tryDelay()$ in Algorithm 14: we try to stretch the tasks one after the others, saving the slow down of the task if it yields a lower energy.

Algorithm 14: $tryDelay()$, version 2

```
forall i ∈ eligible do
    stretch(T_i) ;
    E_i^(ifStr) = computeEnergy();
    if E_i^(ifStr) < E^(est) then
        E^(est) = E_i^(ifStr) ;
        if end_i < t + timeStep then
            schedule(T_i) ;
    else
        rollback();
        schedule(T_i) ;
        update(nextInLine);
```

We illustrate in Figure 6.2 a case in which **LEneS** is better than **SRP**. **SRP** delays T_2 since it is initially the longest path task. **LEneS** does not slow down T_2, because that would prevent all tasks $T_3, \ldots, T_{k+1}$ from using lowest frequencies, hence leading to a higher energy estimation.

On our implementation choices. In the initial paper, the notion of time step is not mentioned: when a task is stretched, its duration gains one "time unit". Firstly this makes the heuristic exponential, and secondly, if all weights and communication times are scaled by the same factor, the algorithm will not return the same result. That is why we introduce the time step, and we have chosen to make 1000 time steps, after some simulation tests.

Another major difference between the algorithm described in the initial paper and the algorithm we implemented is the question of missed deadline. In the initial algorithm, the end time of a task T_i such that $t < LFT_i \leq t + 1$ can be delayed until $t + 1$. However the final deadline cannot be missed. To fix this problem, they include in their priority function (which is in our algorithm only the difference between the energy estimate without and with delay) a term that depends on the length of the critical path from the concerned task. This term is weighted by a new parameter. The lower parameter, the less

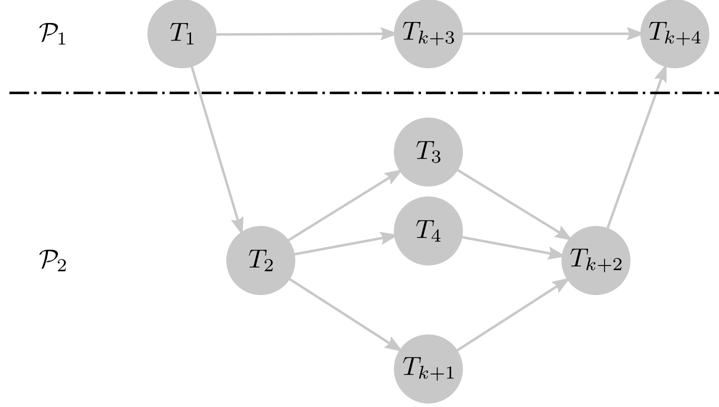


Figure 6.2: Example for **LEneS**.

opportunity for the task to be delayed. The algorithm is launched with a set of initial parameters; if the deadline is missed, those parameters are slightly increased, in order to lower the number of delays. Then the algorithm is launched again, parameters are modified, and so on, until a valid solution is found. We will see in the simulations results that such an algorithm seems to be much slower without saving more energy than our modified variant.

LEneS$_{\text{NO-VDD}}$

We also designed an algorithm for the case where VDD-HOPPING is not allowed. The first change takes place in the estimation of the energy of a task. When VDD-HOPPING is allowed, $t \mapsto E_i(t)$ is a piecewise linear function. Without VDD-HOPPING, we consider that this function is a step function.

In this variant without VDD-HOPPING, we discretize the time in another way: a task T_i has now a discrete set of possible durations, which are namely $w_i/s_{n_f}, \dots, w_i/s_1$. Therefore, at each iteration, we do not consider any more all the tasks whose current end time is between t and $t + \mathit{timeStep}$; we work on the current unscheduled task whose end time is minimum, and compute the energy estimate if this task is run at the highest frequency that is lower than the current frequency. If the new energy is lower than the current one, we lower the frequency, and if not, we schedule the task under the current frequency. The algorithm remains polynomial: each iteration is done in polynomial time, and there are at most $n_f \times n$ iterations.

6.4.5 Opt

The initial paper (see [8]) did not consider communication costs. However, since the mapping onto processors is given, we have easily patched the communications onto the initial work.

Opt$_{\text{VDD}}$

The $a_{i,j}$, for $(i, j) \in \{1, \dots, n\} \times \{1, \dots, n_f\}$, and $start_i$, for $i \in \{1, \dots, n\}$, are rational variables. $a_{i,j}$ is the time during which the processor, that T_i is assigned to, is running at frequency s_j, while, like previously, $start_i$ is the start time of task T_i. We have the following constraints:

– All tasks begin after the time $t = 0$:

$$\forall i \in \{1, \dots, n\}\ ,\ start_i \geq 0$$

– Every task is finished before the deadline:

$$\forall i \in \{1, \dots, n\}\ ,\ start_i + \sum_{j=1}^{n_f} a_{i,j} \leq D$$

– A task must wait until its direct predecessor tasks are done and data is received. For all $(i, i') \in \{1, \dots, n\}^2$, if there is a dependency from T_i to $T_{i'}$:

$$start_i + \sum_{j=1}^{n_f} a_{i,j} + \delta_{i \to i'} \leq start_{i'}$$

– All computations are done:

$$\forall i \in \{1, \dots, n\}\ ,\ \sum_{j=1}^{n_f} a_{i,j} \times s_j \geq w_i$$

Finally, the objective function is given by:

$$\sum_{i=1}^{n} \sum_{j=1}^{n_f} a_{i,j} \times s_j^3$$

Opt$_{\text{NO-VDD}}$

It has been shown, e.g. in [8], that the problem is NP-complete under the NO-VDD-HOPPING model. We have thus an integer linear program, giving the solution in exponential time, by adapting the previous linear program to the NO-VDD-HOPPING strategy. The $a'_{i,j}$, for $(i, j) \in \{1, \dots, n\} \times \{1, \dots, n_f\}$ are binary variables: $a'_{i,j} = 1$ if and only if the task T_i is running at frequency s_j.

– The first constraint is unchanged.

– If the processor that a task T_i is assigned is running at frequency s_j, the task lasts w_i/s_j, hence:

$$\forall i \in \{1, \dots, n\}\ ,\ start_i + \sum_{j=1}^{n_f} a'_{i,j} \times \frac{w_i}{s_j} \leq D$$

– The third constraint is rewritten as the second one: for all $(i, i') \in \{1, \dots, n\}^2$, if there is a dependency from T_i to $T_{i'}$:

$$start_i + \sum_{j=1}^{n_f} a'_{i,j} \times \frac{w_i}{s_j} \leq start_{i'}$$

– The fourth constraint is now useless, but we must ensure that processors choose a speed: for all $i \in \{1, \dots, n\}$,

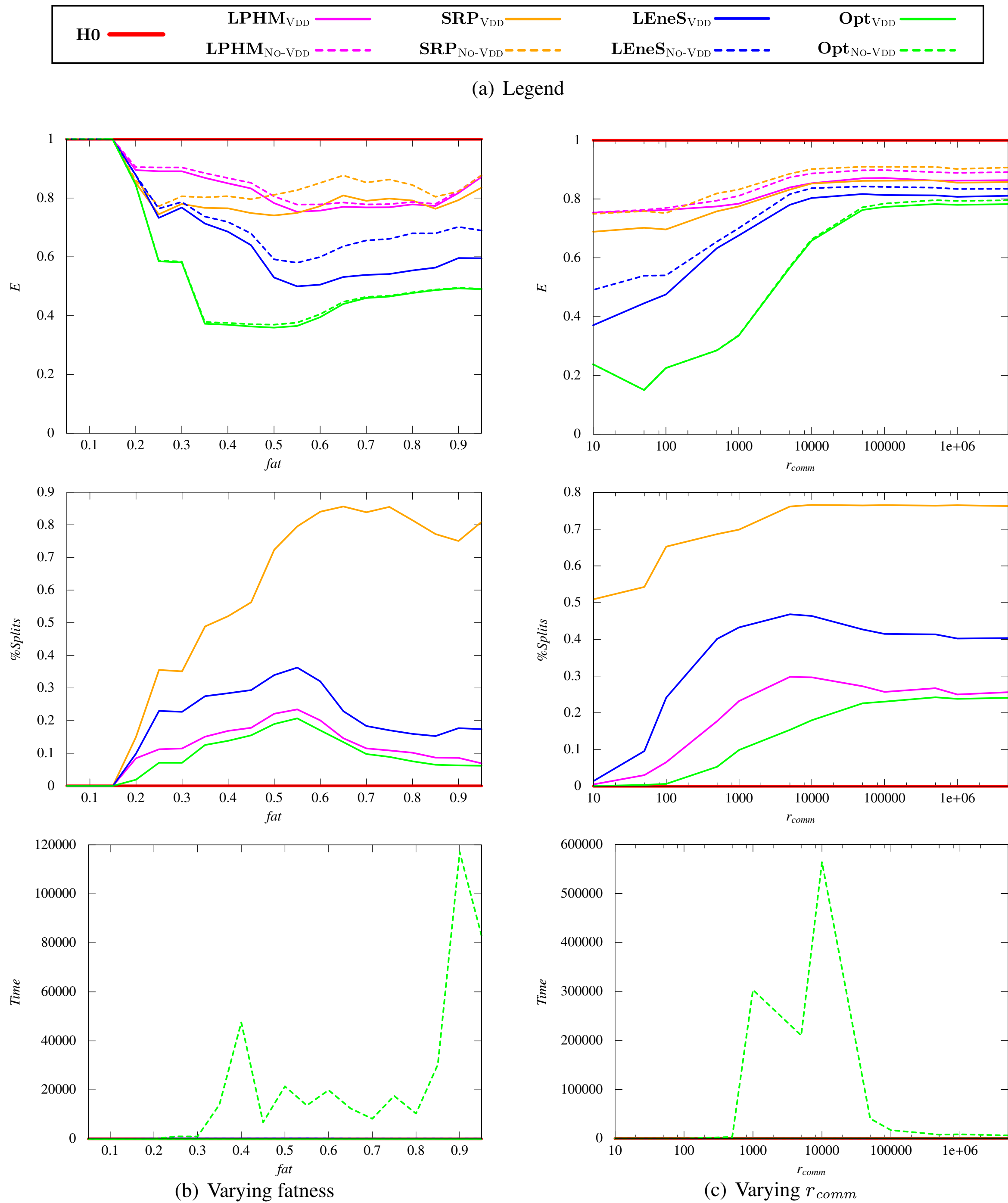
$$\sum_{j=1}^{n_f} a'_{i,j} \geq 1$$

The objective function is slightly different:

$$\sum_{i=1}^{n} \sum_{j=1}^{n_f} a'_{i,j} \times w_i s_j^2$$

6.5 Simulations

We have conducted a large set of simulations, in order to assess the performance of each heuristic, and to measure the contribution of the VDD-HOPPING strategy, with respect to the NO-VDD-HOPPING



(a) Legend

(b) Varying fatness

(c) Varying r_{comm}

Figure 6.3: Fatness and communication-to-computation ratio impact

strategy. We try to exhibit different parameters, which play an important role in the quality of the solutions found by the heuristics. After several tests, we have identified three main characteristics that lead the behavior of the heuristics, which are: the fatness of the DAG, the communication-to-computation ratio, and the number of nodes.

We generate random graphs thanks to the DAG generator Daggen (see [112]). A crucial parameter is the fatness (denoted in the following by fat) of the graph, which is the maximum number of tasks that can be executed concurrently over the total number of tasks. This value ranges from 0 to 1; when fat tends to 0, the graph comes closer to a chain, whereas when $fat = 1$, the graph is a fork-join graph. We draw randomly the weights of the edges and the weights of the vertices between 10^5 and 10^8, then we divide the weights of the edges by a factor r_{comm}. We will vary this parameter in the simulations, from 10 to 10^5. The processor we simulate here has 5 possible speeds: from 1000 operations per second, to 5000, within a step of 1000. Also the communication-to-computation ratio is between 10^{-2} and 10^2. This gives a good view on the heuristics behavior at the border (the simulations we launched with usual ratios ranking from 0.1 to 10 led to intermediate behaviors).

When we plot the energy of the schedule, we normalize the energy of the heuristic solution by the energy consumed if no slack is reclaimed. This heuristic that does not reclaim energy is noted **H0**. We also plot the execution time of the algorithms, which is given in milliseconds. Finally, in order to better understand what is not explicitly given by the energy value, we plot the percentage of tasks that are run at two different frequencies in the solution of the heuristics.

6.5.1 Fatness and Communication-to-computation ratio

We start with 100-node graphs, and study the variations of the heuristics when we increase either the fatness of the graph or the communication-to-computation ratio. In a general manner, the higher the communications, the more energy savings in the optimal solution (see 6.3(b)). This can be explained easily by the fact that there is more variation in the communication costs, hence more slack in the initial schedule, that is reclaimed by **Opt**. In addition we observe that there is less VDD-HOPPING utilization in the case of high communications: there are many tasks that are run at the minimum frequency that cannot be slowed down even more.

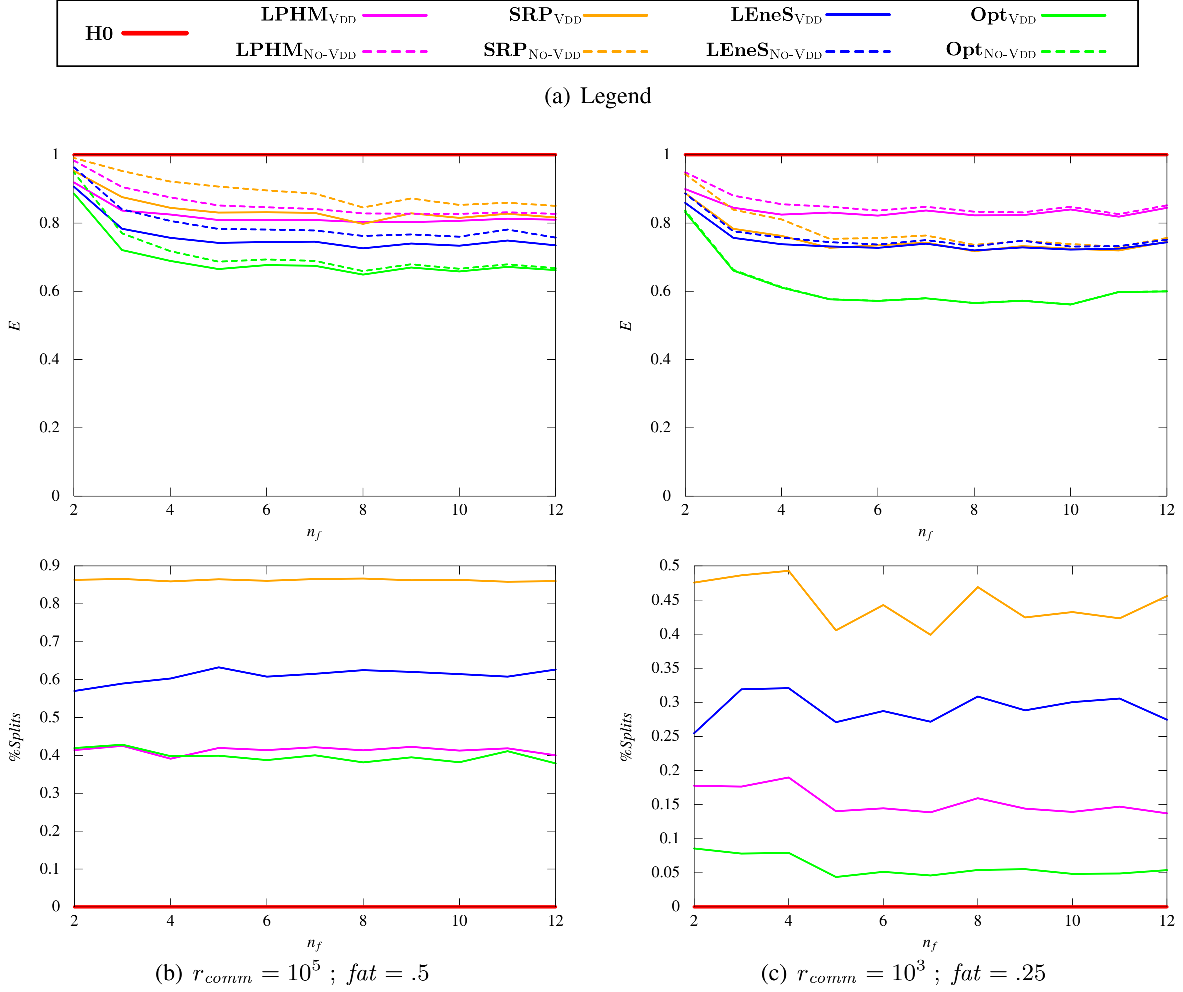
The most surprising observation is that **Opt**$_{\text{NO-VDD}}$ is really close to **Opt**$_{\text{VDD}}$ in most cases. **Opt**$_{\text{NO-VDD}}$ is slightly not as good when computations are very high; otherwise, it can distribute the slack among several tasks, so that almost all of them are given a slot corresponding to a frequency of the set of possible speeds. For example, on Figure 6.3(c), we can see that around 20% of the tasks are split in the solution **Opt**$_{\text{VDD}}$, when $r_{comm} > 50000$. However the total energy curves are almost together.

From a fatness point of view, **SRP** is better when task graphs are willowy, which is in agreement with the example of section 6.4.3: **SRP** can distribute the slack among long paths in a good way. Concerning **LEneS**, energy and percentage of split curves are in the same shape as **Opt**, thanks to its global approach: it can gain high energy when graph fatness and communication costs are above par.

6.5.2 Number of frequencies

We are interested here more particularly in the relevance of the VDD-HOPPING model. We saw in the previous subsection that VDD-HOPPING seems to not improve significantly the energy saving. Now we study how the number of frequencies available on the processors impact the difference between **Opt**$_{\text{VDD}}$ and **Opt**$_{\text{NO-VDD}}$. We keep for the frequencies the same two extreme values (1000 and 5000 operations per second), and vary the number of frequencies from 2 to 12.

We operate those variations for each couple (fat, r_{comm}), where $fat \in \{0.25, 0.5, 0.75\}$ and $r_{comm} \in$



(a) Legend

(b) $r_{comm} = 10^5$; $fat = .5$

(c) $r_{comm} = 10^3$; $fat = .25$

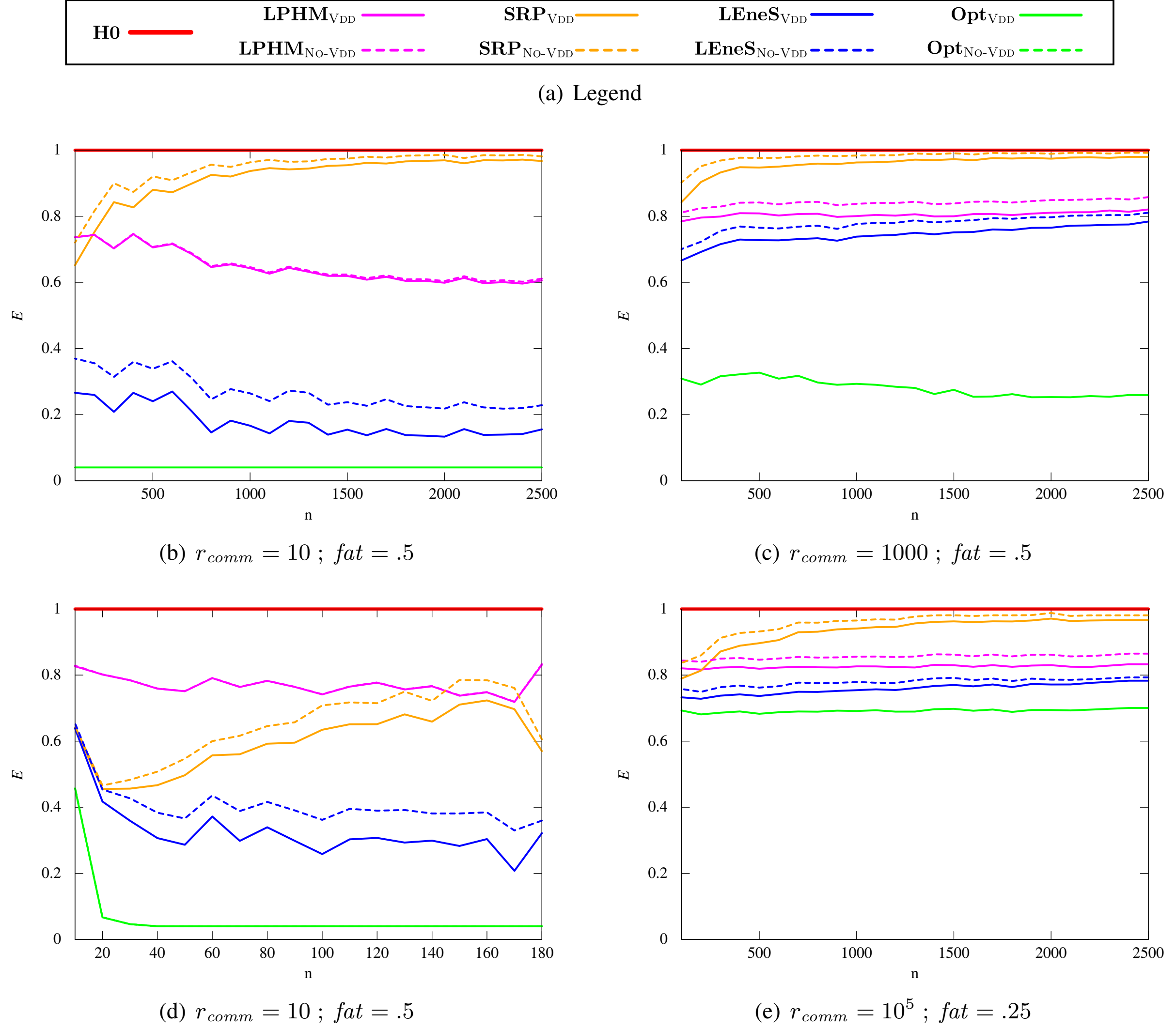
Figure 6.4: Number of frequencies impact

$\{10, 10^3, 10^5\}$. We include only some of the result graphs (on Figure 6.4) in order not to clutter up the discussion.

On Figure 6.4(b) we represent the energy for the couple that leads to the maximum energy difference between **Opt**$_{\text{VDD}}$ and **Opt**$_{\text{No-VDD}}$. With two frequencies, all the heuristics are better than **Opt**$_{\text{No-VDD}}$ in their VDD-HOPPING variants, but **Opt**$_{\text{VDD}}$ gains only around 10% of the energy consumed in **H0**. Note also that a ratio maximum frequency over minimum frequency of 5 is very pessimistic (compared to the current processors) for the NO-VDD-HOPPING variants. Then, for at least three frequencies, **Opt**$_{\text{VDD}}$ achieves from 5 percentage points to 1 percentage point less than **Opt**$_{\text{No-VDD}}$ of the reference energy. Results obtained with other parameters show even less energy saving in **Opt**$_{\text{VDD}}$ compared to **Opt**$_{\text{No-VDD}}$, as in Figure 6.4(c).

6.5.3 Graph size

We pursue in this subsection the general study of the heuristics, and particularly their scalability. It is here about the scalability according to the number of nodes; the scalability on the number of processors is not relevant, since the mapping of the graph is given.



(a) Legend

(b) $r_{comm} = 10$; $fat = .5$

(c) $r_{comm} = 1000$; $fat = .5$

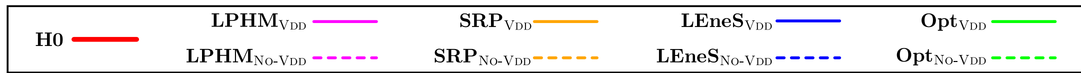
(d) $r_{comm} = 10$; $fat = .5$

(e) $r_{comm} = 10^5$; $fat = .25$

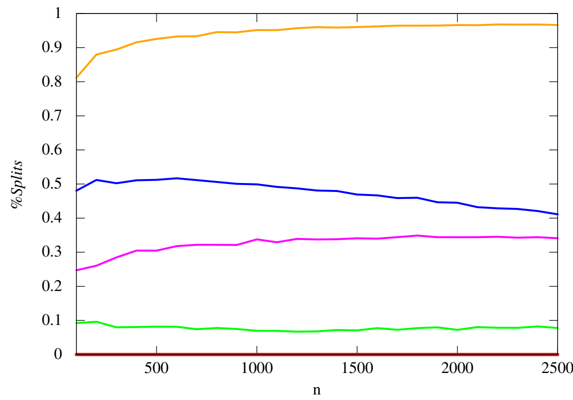
Figure 6.5: Scalability (energy)

We perform two sets of simulations: for graphs whose size is less than 180 nodes, we run all heuristics and linear programs, for both VDD-HOPPING and NO-VDD-HOPPING strategies. Then for graph whose size relies between 100 and 2500, we do not launch any more the linear program with NO-VDD-HOPPING, since it becomes intractable. Like previously, we operate those variations for each couple (fat, r_{comm}), where $fat \in \{0.25, 0.5, 0.75\}$ and $r_{comm} \in \{10, 10^3, 10^5\}$, and we include only some of the result graphs in Figure 6.5 and 6.6.

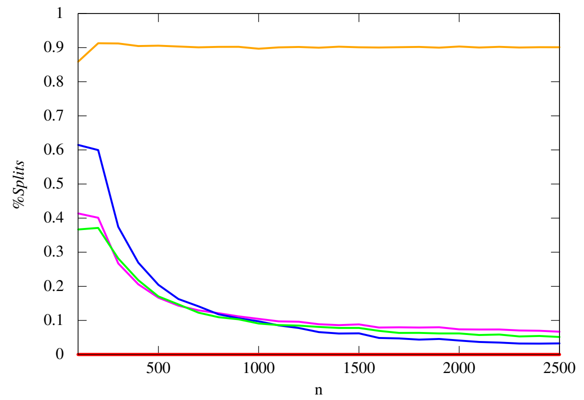
By comparing Figures 6.5(b) and 6.5(c), we can see that **LEneS** comes closer to the optimal solution when communications are high and hence when there is more slack. The global energy estimation can anticipate that consuming the slack will not be prejudicial later in the graph. The idea of **SRP** heuristic seems to work well on small graphs, as we observe on Figure 6.5(d), but on bigger graphs, as we can see on the three other sub-figures of Figure 6.5, it crumbles and renders its slack reclamation completely useless. It appears to be the less scalable heuristic of all heuristics.



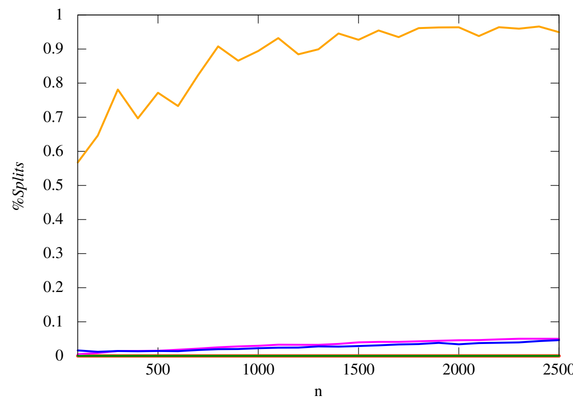
(a) Legend



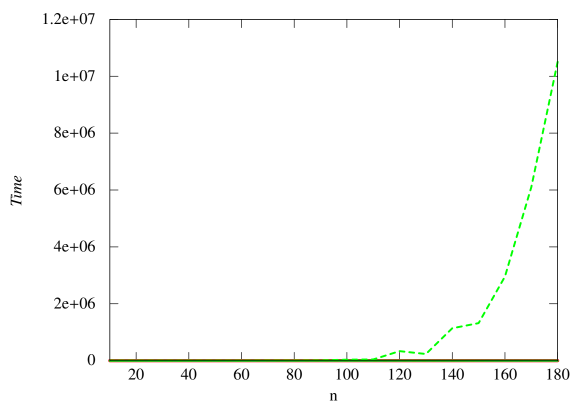
(b) $r_{comm} = 10^3$; $fat = .5$



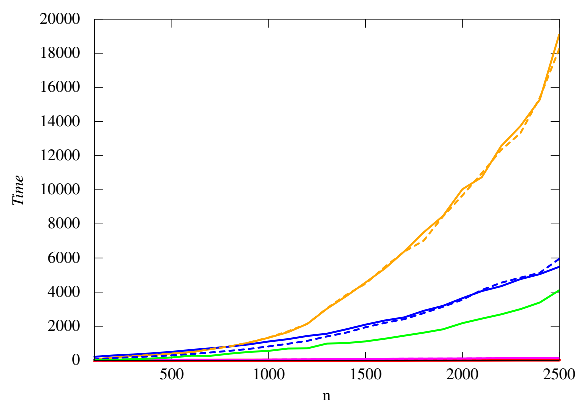
(c) $r_{comm} = 10^5$; $fat = .5$



(d) $r_{comm} = 10$; $fat = .5$



(e) $r_{comm} = 10^5$; $fat = .5$



(f) $r_{comm} = 10^3$; $fat = .25$

Figure 6.6: Scalability (splits and execution times)

Figure 6.5(e), augmented with the previous ones, confirms that the optimal solution benefits drastically from big communications, that creates much slack to reclaim. Finally, those simulations support the fact that despite its naive approach, **LPHM** obtains reasonably good results. It is often better than **SRP**, and performs 20% energy savings in average. That makes it the heuristic with the best value for execution time.

If we take a look at the form of the solutions, we remark on Figure 6.6(b) and 6.6(c) that **SRP** takes full use of VDD-HOPPING strategy, even when all other heuristics do not feel this need. **SRP** gives a tiny part of the slack to plenty of tasks, whereas the other algorithms have better to lower many tasks to their minimum frequency. Those behaviors are stronger when communications get higher (see Figure 6.6(d)).

The solutions given by **Opt** are much better than the solutions found by the heuristics, but what about the execution time? As expected, and as we can see on Figure 6.6(e), $\mathbf{Opt}_{\text{NO-VDD}}$ is very slow, and becomes more or less intractable as soon as the number of nodes exceeds 180 nodes. The execution time of $\mathbf{Opt}_{\text{VDD}}$ is more interesting: on Figure 6.6(f), we observe that it is not slower than **LEneS**, and even faster than **SRP**.

6.5.4 Conclusion of the simulations

In the paper where **LEneS** was described (which achieves 169 citations), authors have claimed that their algorithm saves around 30% of energy, which we confirm in this study. They also have given their algorithm execution time with a graph of 100 nodes, when it is mapped onto 10 processors: $25\,min$. Even with our implementation, for which the algorithm lasts less than $1\,s$, the linear program solves the problem within a smaller time. We can conclude, without any doubt, that if we want to use a slack reclamation algorithm, we have better to use the linear program.

Another contribution of this study is to weight the relevance of VDD-HOPPING. The absolute solution in the VDD-HOPPING strategy, compared to the solution with NO-VDD-HOPPING, is not radically better: those two solutions are even very close. However VDD-HOPPING carries a big interest, since an optimal solution can be found in a small time, which is not the case with NO-VDD-HOPPING.

6.6 Conclusion

In this chapter, we have assessed the performance of several slack reclamation algorithms. We are given a DAG that has already been mapped onto a set of processors, those processors running at their highest frequency. The aim of slack reclamation algorithms is to slow down some tasks, so that the energy consumed is minimized, while the deadline constraint is not violated. We tackle two different models: VDD-HOPPING, where processors can change their frequency at any time and NO-VDD-HOPPING, where a task must be executed at a unique frequency.

We have compared **LPHM**, **SRP**, **LEneS** and **Opt** (described respectively in [14, 71, 50, 8]) through a large set of simulations. **LPHM** is a very fast algorithm that obtains reasonable energy savings in any situation. **SRP** is specialized in fat and small graphs, otherwise it becomes slow and not competitive, while **LEneS** succeeds in saving much energy when it is possible within an interesting time. However, under the VDD-HOPPING model, all those heuristics are useless, since the optimal solution can be obtained through $\mathbf{Opt}_{\text{VDD}}$ with an execution time lower than execution times of **LEneS** and **SRP**. On the contrary, those heuristics are useful when VDD-HOPPING is not allowed, since $\mathbf{Opt}_{\text{NO-VDD}}$ becomes intractable as soon as the graph size is higher than 200.

Simulations have also proved that energy gain in the optimal energy in the VDD-HOPPING model is very close to the energy consumed in the NO-VDD-HOPPING model. As changing frequency leads to overhead on real machines, the sole advantage of using VDD-HOPPING is that we can solve optimally this problem within a small time.

Conclusion

In this thesis, we have studied several topics related to energy-aware scheduling. On the theoretical side, we have classified the problems that we have addressed, and studied the complexity of every problem instance, whereas on the practical side, we have designed many efficient polynomial-time heuristics on general problems that had been shown NP-complete beforehand. We have tried to exploit the potential of a tremendous tool: the Dynamic Voltage and Frequency Scaling (DVFS). From the independent tasks mapped onto independent processors, to the series-parallel task graphs mapped onto a chip multiprocessor, and including the replica placement problem, we have used DVFS to design new scheduling algorithms. Our main contributions are recalled in the following paragraphs.

Summary

On the performance of greedy algorithms for power consumption minimization

In this first chapter, we have revisited the well-known greedy algorithm that aims at minimizing the makespan of independent jobs onto a given set of processors: the jobs are first sorted by non-increasing size in the offline version, and in both versions they are assigned greedily to the currently least loaded processor.

This algorithm fits well with the energy objective, since each assignment is the best local solution from an energy point of view. A lot of papers have been turning around this problem: tight bounds have already been exhibited for close problems, and several papers deal with upper bounds on more general problems. To the best of our knowledge, we are the first to exhibit tight bounds on this problem.

Mapping concurrent streaming applications

In this second chapter, we have performed a comprehensive theoretical study on the mapping of pipelined streaming applications, while designing useful heuristics to obtain reasonable solutions in polynomial time. We have been interested in all possible mono-criterion, bi-criteria and tri-criteria optimization problems in which the period, the latency and the energy were involved. In order to separate the complexity of the problems, we defined different classes of heterogeneity for the platform, and different mapping rules, more or less constrained. We have shown the NP-completeness or exhibited a polynomial-time algorithm for each problem, i.e., a combination of criteria, class of platform and mapping rule. No complexity hole has been left behind. We even generalized the initial problem to another latency model, under which we have derived some new complexity results.

For the tri-criteria problem on a communication homogeneous platform, we have designed several heuristics, as well as an integer linear program giving the optimal solution — which by the way works for any platform. We have launched different simulations, in order to assess the performance, the complexity and the scalability of the heuristics, and they have shown satisfactory results.

Replica placement and update strategies in tree networks

In this third chapter, we have tackled the classical replica placement problem, which we enriched with power saving and dynamicity requirements. Given a distribution of pre-existing servers and a set of requests, servers have to be moved, created or deleted, so that all requests are served, the distribution of the servers stays quite stable (i.e., the cost does not exceed a bound) and the consumed power is minimum.

We have shown three main theoretical results. First, we have written an algorithm in $O(N^5)$, where N is the number of nodes of the tree, for minimizing the cost (without power consideration). Then, we have shown that the problem of minimizing the power (without cost consideration) is NP-complete. Finally, in the case where the number of modes for the servers is bounded, we have designed polynomial-time algorithms that minimize the power under a cost bound. To be sure that those algorithms are useful, we have run convincing simulations showing an important loss, from a cost and power perspective, if those criteria had not been taken into account in the algorithm.

Mapping series-parallel workflows onto chip multiprocessors

In this fourth chapter, we have addressed the problem of mapping streaming applications, which are now in the form of a series-parallel graph, onto a fashion platform: the chip multiprocessor.

We wanted this mapping to be easily implemented, hence we have enforced a secure mapping strategy. The series-parallel graph is first partitioned, so that the graph, whose nodes are partitions of the initial graph and whose edges link two nodes of two different partitions, is a directed acyclic graph. Then the obtained graph is mapped onto the CMP under a one-to-one rule. We have shown a few complexity results, depending on the CMP profile (uni-line or multi-line, uni-directional or bi-directional). Then we have designed various heuristics, some of them directed by polynomial-time algorithms described in the theoretical part. All those heuristics have been tested through simulations, with both randomly generated graphs and real-life ones.

Manhattan routing on chip multiprocessors

In this fifth chapter, we have still considered the chip multiprocessors, but we have centered our approach around the energy consumed by communications. This energy is indeed expected to represent an increasing part of the whole consumed energy in future CMPs.

We have tackled the problem of routing a given set of communications through the CMP. In all current systems communications are routed following an XY route. The robustness of this technique is currently favored, but it also leads to a bad load-balancing between the different links of the CMP, which is becoming critical. That is why we have studied in this chapter how much we could gain from other routing strategies. We have considered Manhattan routings (the communication must follow a Manhattan path from the source core to the destination core), either single-path or multi-path (the communication can be split among several paths). In particular we have shown that the minimum upper bound for the ratio of the power consumed by an XY routing over the power consumed by any Manhattan routing is in $O(p^{\alpha-1})$, where p^2 is the number of cores in the CMP and α is the α of the power formula.

We have enhanced this worst-case analysis with the design of various heuristics finding single-path Manhattan routings. Through simulations, we have confirmed that using Manhattan routings can save huge amounts of energy, and more basically, that it can just help greatly to find a solution, in which the bandwidths of the links are not exceeded.

Assessment of bi-criteria heuristics for general directed acyclic graphs

In this last chapter, we have studied several slack reclamation algorithms found in the literature. The objective of such algorithms is, given a DAG mapped onto a set of processors, to minimize the energy consumption without increasing the initial execution time. We have also quantified the difference of energy savings between the two following execution models: VDD-HOPPING, where processors can change their frequency in the middle of a task execution, and NO-VDD-HOPPING, where processors must wait for an inter-task interval to operate the frequency modification.

We have shown that in the NO-VDD-HOPPING model, the linear program that gives the optimal solution is faster than most of the heuristics while yielding much more energy saving. This fact leads us to discard the use of the other heuristics we studied. We have also assessed the contribution of the VDD-HOPPING model: while the gain from an energy perspective is very low, finding the optimal solution in this model is remarkably fast. This is to be contrasted with the NO-VDD-HOPPING variant which is intractable for graphs with around 150 nodes.

Perspectives

We first outline some extensions related to the results obtained in the previous chapters. Then we state more general, long-term oriented, research directions.

Mapping concurrent streaming applications

First of all, we did not try to map the applications onto other platforms than processor cliques. It may be interesting to study the impact on the theoretical results if the applications are now mapped onto a chip multi-processor, for instance. The grid structure of the CMP seems to fit well with the linearity of the pipelined applications. In the same way, we could find new heuristics that would take advantage of this matching between the configuration of the applications and the architecture of the platform.

We have restricted ourselves to map each task onto a single processor for the ease of implementation; however, the replication could turn out to be very efficient. If two processors are assigned to the same task, each of them handling successively half of the data sets in a round-robin fashion, the period of the application can potentially be halved. On the other side, the energy consumed is doubled. But we could have looked from the reverse point of view: we could keep the same period, and halve the frequency of both processors, which leads to a power reduction of $2^{\alpha-1}$. A new interesting trade-off appears, because of the replication, and deserves further studies.

Replica placement and update strategies in tree networks

We have proved the NP-completeness of the power minimization problem only when the frequencies are discrete. The generalization to the case of continuous frequencies is not trivial at all, and it would be really interesting to find the complexity of this close problem.

On the practical side, the algorithms described in this chapter are optimal, but have a high-complexity that renders them intractable for large problem instances. It would be appropriate to break this barrier, and to find non-optimal, though competitive, algorithms that would be efficient in terms of execution time.

The replica placement problem can be declined in numerous variants. For instance, some work has been done when a minimum quality of service must be ensured. The edges in the tree are weighted, and represent a distance; a maximum distance between a client and its server should not be exceeded. We

have studied approximation algorithms for this variant of the problem and obtained convincing results (see [B2]). Therefore we envision to derive approximation algorithms on the power-aware version of replica placement problem.

Mapping series-parallel workflows onto chip multiprocessors

In this chapter, we have only considered mappings with DAG-partitions. It may be exciting to think about what can be done with other mapping rules. With the DAG-partition mapping rule, the problem becomes NP-complete, as soon as the CMP is not uni-line, uni-directional and the elevation of the series-parallel graph is not bounded. We could define a more restrictive mapping rule, composed of linear chains for instance, and enlarge the class of CMPs in which the problem is polynomial. Or we could study general mappings, and try to solve the numerous problems that would arise, e.g., buffer sizes and deadlocks.

One can consider that mapping streaming applications onto a single CMP is unduly restrictive. Therefore we could generalize our algorithms to groups of heterogeneous CMPs. By the way we could take this opportunity to refine the CMP model that we have been used. With the increasing number of cores in the multi-cores, which are thus often called many-cores, those cores will not be supplied by the same voltage. There will be islands of cores, in which cores will share the same supply voltage. This new level of heterogeneity calls for new complexity results, and more complicated algorithms.

Manhattan routing on chip multiprocessors

On the short term, we would like to close the worst-case analysis by finding the minimum upper bound of the ratio of the power consumed by an XY routing over the power consumed by a single-path Manhattan routing when the communications go from the same source node to the same destination node.

On the longer term, we envision to deal with the optimal solution of Manhattan routings. It may be interesting to obtain this optimal solution, through an optimized integer linear program, for example, even for small instances. This would allow us to have a better idea about the performance of the heuristics. A more challenging future work consists in finding approximation algorithms on this Manhattan routing problem, even if it does not seem easy. Finally, we would like to design new heuristics for the multi-path problem; new ideas need to be found, but some of the heuristics for the single-path problem can also be easily converted.

Assessment of bi-criteria heuristics for general directed acyclic graphs

We point out that the model cannot take all parameters into account; overheads in time occur when a processor is upgraded or downgraded, which can lead to a larger makespan, the temperature of a processor that is never idle can stay high and elevate the energy consumption, etc. It would be thus interesting to push the simulations to real experiments. That would give us more information about the quality of the model, the real-life performance of the algorithms that we implemented, and the utility of the VDD-HOPPING strategy.

General perspectives

The first natural direction is to generalize the energy consumption model. In this thesis, we have only considered the energy consumed by a processor; this has included the energy consumed to compute and

the energy consumed to communicate. In the future, we could take the disk drive energy consumption into account, and for instance shut it down when it is unused.

Concerning the Dynamic Voltage and Frequency Scaling model, we could improve the model that we have been using and in which we can upgrade or downgrade a core without overhead. In reality, a little delay and an energy consumption overhead occur when the processor frequency is modified, depending on which frequencies are brought into play. Those delays, in the order of $100\,\mu s$, and those energy overheads, in the order of $1\,mJ$ with current processors, can help us improve the accuracy of the model, and lead to interesting problems.

We saw that minimizing the energy consumption is required to deliver a functional Exascale machine. In such a machine the reliability is another key-criterion that we need to be interested in, because of the multiplication of the number of processors. A combination of those two criteria may lead to challenging problems in numerous cases.

The temperature plays non-essential though significant role in the energy consumption of a whole computer: on the one hand it increases the leakage energy, and on the other hand fans have to consume more energy in order to maintain the temperature within a reasonable level. Tackling temperature-related problems, even though that can lead to completely different solution forms, can thus indirectly help to reduce the energy consumption.

Bibliography

[1] Kunal Agrawal, Anne Benoit, Loic Magnan, and Yves Robert. Scheduling algorithms for workflow optimization. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 201–212. IEEE Computer Society, 2010.

[2] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jórg Henkel. ADAM: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the Design Automation Conference (DAC)*, pages 760–765. ACM, 2008.

[3] Noga Alon, Yossi Azar, Gerhard J. Woeginger, and Tal Yadid. Approximation schemes for scheduling. In *Proceedings of the ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 493–500. Society for Industrial and Applied Mathematics, 1997.

[4] Marina Alonso, Salvador Coll, Juan-Miguel Martínez, Vicente Santonja, Pedro López, and José Duato. Dynamic power saving in fat-tree interconnection networks using on/off links. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 299–309. IEEE Computer Society, 2006.

[5] AMD. ACP - The Truth About Power Consumption Starts Here. `http://www.amd.com/us/Documents/ 43761C_ACP_WP_EE.pdf`, 2010.

[6] Alexandru Andrei, Marcus Schmitz, Petru Eles, Zebo Peng, and Bashir M. Al-Hashimi. Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, volume 1, pages 518–523. IEEE Computer Society, 2004.

[7] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. Multi-objective mapping for mesh-based noc architectures. In *Proceedings of the International Conference on Hardware/ Software Codesign and System Synthesis (CODES+ISSS)*, pages 182–187. ACM, 2004.

[8] Guillaume Aupy, Anne Benoit, Fanny Dufossé, and Yves Robert. Reclaiming the energy of a schedule, models and algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–136. ACM, 2011.

[9] Guillaume Aupy, Anne Benoit, and Yves Robert. Energy-aware scheduling under reliability and makespan constraints. Research report RR-7757, INRIA, 2012.

[10] Adi Avidor, Yossi Azar, and Jiří Sgall. Ancient and new algorithms for load balancing in the lp norm. In *Proceedings of the ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 426–435. Society for Industrial and Applied Mathematics, 1998.

[11] Baruch Awerbuch, Yossi Azar, Edward F. Grove, Ming yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Load balancing in the lp norm. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 383–391. IEEE Computer Society, 1995.

[12] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 113–121. IEEE Computer Society, 2003.

[13] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1–39, 2007.

[14] Sanjeev Baskiyar and Kiran Kumar Palli. Low power scheduling of dags to minimize finish times. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC)*, pages 353–362. Springer Verlag, 2006.

[15] Michael A. Bender, Soumen Chakrabarti, and Muthu S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 270–279. Society for Industrial and Applied Mathematics, 1998.

[16] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Replica placement and access policies in tree networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(12):1614–1627, 2008.

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

[18] Anne Benoit and Yves Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 57(4):689–724, August 2010.

[19] Anne Benoit, Yves Robert, and Eric Thierry. On the complexity of mapping linear chain applications onto heterogeneous platforms. *Parallel Processing Letters (PPL)*, 19(3):383–397, 2009.

[20] Peter Blaha, Karlheinz Schwarz, Georg Madsen, Dieter Kvasnicka, and Joachim Luitz. WIEN2k: An Augmented Plane Wave Plus Local Orbitals Programfor Calculating Crystal Properties - User's guide, 2001. Vienna University of Technology, Austria.

[21] Geoffrey Blake, Ronald G. Dreslinski, and Trevor N. Mudge. A survey of multicore processors. *Signal Processing Magazine*, 26(6):26–37, 2009.

[22] David P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 190–196. ACM, 2006.

[23] Ashok K. Chandra and Chak-Kuen Wong. Worst-case analysis of a placement algorithm related to storage allocation. *SIAM Journal on Computing (SICOMP)*, 4(3):249–263, 1975.

[24] Anantha P. Chandrakasan and Amit Sinha. JouleTrack: A Web Based Tool for Software Energy Profiling. In *Proceedings of the Design Automation Conference (DAC)*, pages 220–225. IEEE Computer Society, 2001.

[25] Guangyu Chen, Feihui Li, Mahmut Kandemir, and Mary Jane Irwin. Reducing NoC energy consumption through compiler-directed channel voltage scaling. *Notices of Special Interest Group on Programming Languages (SIGPLAN)*, 41:193–203, 2006.

[26] Guangyu Chen, Feihui Li, Seung Woo Son, and Mahmut Kandemir. Application mapping for chip multiprocessors. In *Proceedings of the Design Automation Conference (DAC)*, pages 620–625. ACM, 2008.

[27] Guangyu Chen, Konrad Malkowski, Mahmut T. Kandemir, and Padma Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 231–242. IEEE Computer Society, 2005.

[28] Jian-Jia Chen. Expected energy consumption minimization in DVS systems with discrete frequencies. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1720–1725. ACM, 2008.

[29] Jian-Jia Chen and Chin-Fu Kuo. Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 28–38. IEEE Computer Society, 2007.

[30] Jian-Jia Chen and Tei-Wei Kuo. Multiprocessor energy-efficient scheduling for real-time tasks. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 13–20. IEEE Computer Society, 2005.

[31] Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 289–294. ACM, 2007.

[32] Jian-Jia Chen and Lothar Thiele. Energy-efficient task partition for periodic real-time tasks on platforms with dual processing elements. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, pages 161–168. IEEE Computer Society, 2008.

[33] Yan Chen, Randy H. Katz, and John Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 306–318. Springer Verlag, 2002.

[34] Zeshan Chishti, Michael D. Powell, and Tomas N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 357–368, 2005.

[35] Sangyeun Cho and Rami G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21:342–353, 2010.

[36] Israel Cidon, Shay Kutten, and Ran Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[37] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[38] Cplex. ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`, -.

[39] DataCutter. DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. `http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm`.

[40] Pepijn de Langen and Ben Juurlink. Leakage-aware multiprocessor scheduling. *Journal of Signal Processing Systems*, 57(1):73–88, 2009.

[41] Jack Dongarra and Pete Beckman. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 2011.

[42] José Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 4:1320–1331, 1993.

[43] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2000.

[44] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[45] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proceedings of the ACM/IEEE conference on SuperComputing (SC)*, pages 34–45. IEEE Computer Society, 2005.

[46] Ronald Lewis Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[47] Luis Gravano, Gustavo D. Pifarré, Pablo E. Berman, and Jorge L. C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 5(12):1233–1251, December 1994.

[48] Greenpeace. Make it green. 2010.

[49] Philippe Grosse, Yves Durand, and Paul Feautrier. Methods for power optimization in SOC-based data flow systems. *ACM Transactions on Design Automation of Electronic Systems*, 14:1–20, 2009.

[50] Flavius Gruian and Krzysztof Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, pages 449–455. ACM, 2001.

[51] Yi Gu and Qishi Wu. Maximizing workflow throughput for streaming applications in distributed environments. In *Proceedings of the International Conference on Computer Communication Networks (ICCCN)*, pages 1–6. IEEE Computer Society, 2010.

[52] Klaus D. Gunther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communications*, 29(4):512–524, 1981.

[53] Mohammad Hammoud, Sangyeun Cho, and Rami Melhem. ACM: An Efficient Approach for Managing Shared Caches in Chip Multi- processors. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 355–372. Springer Verlag, 2009.

[54] Mohammad Hammoud, Sangyeun Cho, and Rami G. Melhem. A dynamic pressure-aware associative placement strategy for large scale chip multiprocessors. *Computer Architecture Letters*, 9(1):29–32, 2010.

[55] Stephen L. Hary and Füsun Özgüner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 10(8):838–851, 1999.

[56] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, 2007.

[57] Yoshihiko Hotta, Mitsuhisa Sato, Hideaki Kimura, Satoshi Matsuoka, Taisuke Boku, and Daisuke Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 340–351. IEEE Computer Society, 2006.

[58] Tai-Yi Huang, Yu-Che Tsai, and Edward T.-H. Chu. A near-optimal solution for the heterogeneous multi-processor single-level voltage setup problem. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 57–68. IEEE Computer Society, 2007.

[59] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(8):1028–1040, 2007.

[60] Intel XScale technology. `http://www.intel.com/design/intelxscale`.

[61] International energy agency. *Gadgets and Gigawatts; Policies for Energy Efficient Electronics*. IEA publications, 2009.

[62] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202. ACM, 1998.

[63] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202. ACM, 1998.

[64] Wooyoung Jang and David Z. Pan. A3MAP: Architecture-Aware Analytic Mapping for Networks-on-Chip. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, pages 523–528. ACM, 2010.

[65] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference (DAC)*, pages 275–280. ACM, 2004.

[66] Konstantinos Kalpakis, Koustuv Dasgupta, and Ouri Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12(6):628–637, 2001.

[67] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[68] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGOPS Operating Systems Review*, 36:211–222, 2002.

[69] Jaeha Kim and Mark A. Horowitz. Adaptive supply serial links with sub-1V operation and per-pin clock recovery. In *Proceedings of the International Solid-State Circuits Conference*, pages 1403–1413. IEEE Computer Society, 2002.

[70] Kyong Hoon Kim, Rajkumar Buyya, and Jong Kim. Power Aware Scheduling of Bag-of-Tasks Applications with Deadline Constraints on DVS-enabled Clusters. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 541–548. IEEE Computer Society, 2007.

[71] Hideaki Kimura, Mitsuhisa Sato, Yoshihiko Hotta, Taisuke Boku, and Daisuke Takahashi. Empirical study on reducing energy of parallel programs using slack reclamation by dvfs in a power-scalable high performance cluster. In *Proceedings of the European Cluster Conference*, pages 21–30. IEEE Computer Society, 2006.

[72] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-aware deadlock-free oblivious routing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 208–219. ACM, 2009.

[73] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[74] Kanishka Lahiri, Anand Raghunathan, Sujit Dey, and Debashis Panigrahi. Battery-driven system design: a new frontier in low power design. In *Proceedings of the Design Automation Conference (DAC)*, pages 261–267. ACM, 2002.

[75] Chun-Yi Lee and Niraj K. Jha. FinFET-based dynamic power management of on-chip interconnection networks through adaptive back-gate biasing. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 350–357. IEEE Computer Society, 2009.

[76] Seongsoo Lee and Takayasu Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the Design Automation Conference (DAC)*, pages 806–809. ACM, 2000.

[77] Seung Eun Lee and Nader Bagherzadeh. A variable frequency link for a power-aware network-on-chip (NoC). *Integration*, pages 479–485, 2009.

[78] Young Choon Lee and Albert Y. Zomaya. On effective slack reclamation in task scheduling for energy reduction. *Journal of Information Processing Systems (JIPS)*, 5(4):175–186, 2009.

[79] Joseph Y.-T. Leung and W.-D. Wei. Tighter bounds on a heuristic for a partition problem. *Information Processing Letters*, 56, 1995.

[80] Feihui Li, Guilin Chen, and Mahmut T. Kandemir. Compiler-directed voltage scaling on communication links for reducing power consumption. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 456–460. ACM, 2005.

[81] Jian Li, Wei Huang, Charles Lefurgy, Lixin Zhang, Wolfgang E. Denzel, Richard R. Treumann, and Kun Wang. Power shifting in thrifty interconnection network. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 156–167. IEEE Computer Society, 2011.

[82] Pangfeng Liu, Yi-Fang Lin, and Jan-Jan Wu. Optimal placement of replicas in data grid environments with locality assurance. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, pages 465–474. IEEE Computer Society, 2006.

[83] Philipp Mahr, Christian Lörchner, Harold Ishebabi, and Christophe Bobda. SoC- MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 187–192. IEEE Circuit and Systems Society, 2008.

[84] Richard McClatchey, Florida Estrella, Jean-Marie Le Goff, Zsolt Kovacs, and Nigel Baker. Object databases in a distributed scientific workflow application. In *Proceedings of the Basque International Workshop on Information Technology (BIWIT)*, pages 11–21. IEEE Computer Society, 1997.

[85] Nithin Michael, Milen Nikolov, Ao Tang, G. Edward Suh, and Christopher Batten. Analysis of application-aware on-chip routing under traffic uncertainty. In *Proceedings of the ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 9–16. IEEE Computer Society, 2011.

[86] Mark P. Mills. The internet begins with coal. *The Greening Earth Society*, 1999.

[87] Ramesh Mishra, Namrata Rastogi, Dakai Zhu, Daniel Mossé, and Rami Melhem. Energy aware scheduling for distributed real-time systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 21–29. IEEE Computer Society, 2003.

[88] Alon Naveh, Efraim Rotem, Avi Mendelson, Simcha Gochman, Rajshree Chabukswar, Karthik Krishnan, and Arun Kumar. Power and Thermal Management in the Intel CoreTM Duo Processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[89] Linwei Niu. Energy Efficient Scheduling for Real-Time Embedded Systems with QoS Guarantee. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 163–172. IEEE Computer Society, 2010.

[90] Takanori Okuma, Hiroto Yasuura, and Tohru Ishihara. Software energy reduction techniques for variable-voltage processors. *Design Test of Computers, IEEE*, 18(2):31–41, 2001.

[91] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *Notices of Special Interest Group on Programming Languages (SIGPLAN)*, 31:2–11, 1996.

[92] The Climate Group on behalf of the Global eSustainability Initiative (GeSI). Smart 2020: Enabling the low carbon economy in the information age. 2008.

[93] John D. Owens, William J. Dally, Ron Ho, D. N. (Jay) Jayasimha, Stephen W. Keckler, and Li-Shiuan Peh. Research Challenges for On-Chip Interconnection Networks. *IEEE Micro*, 27:96–108, 2007.

[94] Rajesh Babu Prathipati. Energy efficient scheduling techniques for real-time embedded systems. Master's thesis, Texas A&M University, 2004.

[95] Kirk Pruhs, Rob van Stee, and Patchrawat Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43:67–80, 2008.

[96] Jun Qin and Thomas Fahringer. Advanced data flow support for scientific grid workflow applications. In *Proceedings of the ACM/IEEE conference on SuperComputing (SC)*, pages 1–12. IEEE Computer Society, 2007.

[97] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[98] Rashedur M. Rahman, Ken Barker, and Reda Alhajj. Effective dynamic replica maintenance algorithm for the grid environment. In *Advances in Grid and Pervasive Computing*, volume 3947, pages 336–345. Springer LNCS 3947, 2006.

[99] Rashedur M. Rahman, Ken Barker, and Reda Alhajj. Replica placement design with static optimality and dynamic maintainability. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 434–437. IEEE Computer Society, 2006.

[100] Paul Renaud-Goud. Source code for the simulations (replicas). `http://graal.ens-lyon.fr/~prenaud/replicas/`.

[101] Paul Renaud-Goud. Source code for the simulations (routing). `http://graal.ens-lyon.fr/~prenaud/Routing/`.

[102] Paul Renaud-Goud. Source Code for the Experiments (CMPs), 2011. `http://graal.ens-lyon.fr/~prenaud/sp-cmp/`.

[103] Felix Schueller, Jun Qin, Farrukh Nadeem, Radu Prodan, Thomas Fahringer, and Georg Mayr. Performance, Scalability and Quality of the Meteorological Grid Workflow MeteoAG. In *Proceedings of the Austrian Grid Symposium*, pages 20–27. OCG Verlag, 2006.

[104] Daeho Seo, Akif Ali, Won-Taek Lim, and N. Rafique. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 432–443. ACM, 2005.

[105] Li Shang, Li-Shiuan Peh, and Niraj K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 91–102. IEEE Computer Society, 2003.

[106] Dongkun Shin. Power-aware communication optimization for networks-on-chips with voltage scalable links. In *Proceedings of the International Conference on Hardware/ Software Codesign and System Synthesis (CODES+ISSS)*, pages 170–175. ACM, 2004.

[107] Mohammad Shorfuzzaman, Peter Graham, and Rasit Eskicioglu. Adaptive popularity-driven replica placement in hierarchical data grids. *Journal of Supercomputing*, 51(3):374–392, 2010.

[108] Laura Silva, Gian Granato, Alessandro Bressan, Cedric Lacey, Carlton Baugh, Shaun Cole, and Carlos Frenk. Modelling dust in galactic seds: Application to semi-analytical galaxy formation models. *Astrophysics and Space Science*, 276:1073–1078, 2001.

[109] Streamit project. http://groups.csail.mit.edu/cag/streamit/apps/stream-graphs, -.

[110] Jaspal Subhlok and Gary Vondran. Optimal mapping of sequences of data parallel tasks. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 134–143. ACM, 1995.

[111] Jaspal Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 62–71. ACM, 1996.

[112] Frédéric Suter. Source Code of Daggen, 1998. http://www.loria.fr/~suter/dags.html.

[113] Kenjiro Taura and Andrew Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Proceedings of the Heterogenous Computing Workshop*, pages 102–115. IEEE Computer Society, 2000.

[114] Haluk Topcuoglu and Min you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13:260–274, 2002.

[115] Girish Varatkar and Radu Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 510–521. IEEE Computer Society, 2003.

[116] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37:195–237, 2005.

[117] Naga Vydyanathan, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Proceedings of the Euro-par conference*, pages 173–183. Springer Verlag, 2007.

[118] Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 254–261. IEEE Computer Society, 2008.

[119] Lizhe Wang, Gregor von Laszewski, Jai Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 368–377. IEEE Computer Society, 2010.

[120] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 368–377. IEEE Computer Society, 2010.

[121] Jan-Jan Wu, Yi-Fang Lin, and Pangfeng Liu. Optimal replica placement in hierarchical Data Grids with locality assurance. *Journal of Parallel and Distributed Computing (JPDC)*, 68(12):1517–1538, 2008.

[122] Qishi Wu, Jinzhu Gao, Mengxia Zhu, Nageswara S. V. Rao, Jian Huang, and S. Sitharama Iyengar. On optimal resource utilization for distributed remote visualization. *IEEE Transactions on Computers (TC)*, 57(1):55–68, 2008.

[123] Qishi Wu and Yi Gu. Supporting distributed application workflows in heterogeneous computing environments. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, pages 3–10. IEEE Computer Society, 2008.

[124] Ruibin Xu, Rami Melhem, and Daniel Mossé. Energy-aware scheduling for streaming applications on chip multiprocessors. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, pages 25–38. IEEE Computer Society, 2007.

[125] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy in real-time embedded systems. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, pages 251–254. ACM, 2005.

[126] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Transactions on Computer Systems (TOCS)*, 25(4):9–17, 2007.

[127] Liu Yang and Lin Man. On-Line and Off-Line DVS for Fixed Priority with Preemption Threshold Scheduling. In *Proceedings of the International Conference on Embedded Software and Systems (ICESS)*, pages 273–280. IEEE Computer Society, 2009.

[128] Yumin Zhang, Xiaobo Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the Design Automation Conference (DAC)*, pages 183–188. IEEE Computer Society, 2002.

[129] Yong Zhao, Michael Wilde, Ian Foster, Jens Voeckler, Thomas Jordan, Elizabeth Quigg, and James Dobson. Grid middleware services for virtual data discovery, composition, and integration. In *Proceedings of the workshop on Middleware for Grid Computing (MGC)*, pages 57–62. ACM, 2004.

[130] Dakai Zhu, Rami G. Melhem, and Bruce R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(7):686–700, 2003.

[131] Yifan Zhu and Frank Mueller. Dvsleak: combining leakage reduction and voltage scaling in feedback edf scheduling. In *Proceedings of the conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 31–40. ACM, 2007.

Publications

Articles in international refereed journals

[A1] Anne Benoit, Paul Renaud-Goud, and Yves Robert. Models and complexity results for performance and energy optimization of concurrent streaming applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(3):261–273, 2011.

Articles in international refereed conferences

[B1] A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert, "Power-aware manhattan routing on chip multiprocessors," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[B2] A. Benoit, H. Larchevêque, and P. Renaud-Goud, "Optimal algorithm and approximation algorithms for replica placement with distance constraints in tree networks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[B3] A. Benoit, P. Renaud-Goud, Y. Robert, and R. G. Melhem, "Energy-aware mappings of series-parallel workflows onto chip multiprocessors," in *Proceedings of International Conference on Parallel Processing (ICPP)*, pp. 472–481, 2011.

[B4] A. Benoit, P. Renaud-Goud, and Y. Robert, "On the performance of greedy algorithms for power consumption minimization," in *Proceedings of International Conference on Parallel Processing (ICPP)*, pp. 454–463, 2011.

[B5] A. Benoit, P. Renaud-Goud, and Y. Robert, "Power-aware replica placement and update strategies in tree networks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 2–13, 2011.

[B6] A. Benoit, P. Renaud-Goud, and Y. Robert, "Sharing resources for performance and energy optimization of concurrent streaming applications," in *SBAC-PAD*, pp. 79–86, 2010.

[B7] A. Benoit, P. Renaud-Goud, and Y. Robert, "Performance and energy optimization of concurrent pipelined applications," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–12, 2010.

Research reports

[C1] A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert, "Power-aware Manhattan routing on chip multiprocessors," Research Report INRIA-RR-7752, Oct. 2011.

[C2] A. Benoit, H. Larchevêque, and P. Renaud-Goud, "Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks," Research Report INRIA-RR-7750, Sept. 2011.

[C3] A. Benoit, P. Renaud-Goud, and Y. Robert, “Models and complexity results for performance and energy optimization of concurrent streaming applications,” Research Report INRIA-RR-7589, Apr. 2011.

[C4] A. Benoit, R. Melhem, P. Renaud-Goud, and Y. Robert, “Energy-aware mappings of series-parallel workflows onto chip multiprocessors,” Research Report INRIA-RR-7521, Apr. 2011.

[C5] A. Benoit, P. Renaud-Goud, and Y. Robert, “Power-aware replica placement and update strategies in tree networks,” Research Report LIP-RR-2010-29, Oct. 2010.

[C6] A. Benoit, P. Renaud-Goud, and Y. Robert, “On the performance of greedy algorithms for energy minimization,” Research Report LIP-RR-2010-27, Sept. 2010.

[C7] A. Benoit, P. Renaud-Goud, and Y. Robert, “Performance and energy optimization of concurrent pipelined applications,” Research Report LIP-RR-2009-27, Sept. 2010.

[C8] A. Benoit, P. Renaud-Goud, and Y. Robert, “Sharing resources for performance and energy optimization of concurrent streaming applications,” Research Report LIP-RR-2010-05, Feb. 2010.