Communication-aware scheduling on heterogeneous master-worker platforms

Jean-François Pineau

N° d'ordre : 475 *N° attribué par la bibliothèque : 07ENSL0475*

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l' École doctorale de Mathématiques et Informatique fondamentale

présentée et soutenue publiquement le 25 septembre 2008 par

Monsieur Jean-François PINEAU

Communication-aware scheduling on heterogeneous master-worker platforms

Directeur de thèse :	Monsieur	Yves	ROBERT	
Co-encadrant de thèse :	Monsieur	Frédéric	VIVIEN	
Après avis de :	Monsieur	Oliver	SINNEN	Membre/Rapporteur
	Monsieur	Denis	TRYSTRAM	Membre/Rapporteur

Devant la commission d'examen formée de :

Monsieur	Emmanuel	JEANNOT	Membre
Monsieur	Jean-Claude	KÖNIG	Membre
Monsieur	Yves	ROBERT	Membre
Monsieur	Oliver	SINNEN	Membre/Rapporteur
Monsieur	Denis	TRYSTRAM	Membre/Rapporteur
Monsieur	Frédéric	VIVIEN	Membre

Merci !

Au cours de ces années de thèse, passées au sein de l'équipe GRAAL, j'ai eu l'occasion, malgré mon tempérament de grizzly ermite hibernant, de rencontrer de nombreuses personnes incroyablement sympatiques et généreuses. Je profite donc de cette section pour montrer que leurs actions ne sont pas passées inaperçues.

Merci à toi, Yves, pour avoir toujours montré le côté fun de l'algorithmique et de l'ordonnancement, dès les premiers cours de L3. Merci donc pour m'avoir donné goût à ces domaines, pour m'avoir aiguillé vers Henri Casanova lors de mon stage de M1, pour avoir toujours été à l'écoute de mes questionnements, pour ne pas avoir perdu espoir lorsque mes expériences de produit de matrices piétinaient, et enfin pour toutes tes "jokes" tout au long de ces 3 (4 ?) années.

Merci à toi, Fredo, pour ta grande empathie, pour avoir voulu revivre en même temps que moi les joies de la rédaction, en écrivant ton HDR peu de temps avant ma rédaction du manuscript. Tu as même voulu partager la pression que j'avais alors que je corrigeais le manuscrit à quelques jours de devoir le rendre, en combinant la relecture de mon manuscript, la soumission de 2 articles, et un déménagement à Hawaii. Pour tout cela, merci. Mais il ne faut pas que j'omette non plus ta rigueur sans faille tout au long de ces années. Merci de ne pas avoir perdu espoir de me transmettre un soupçon de ta "pointillosité", et de ne pas avoir hésité à user tes précieux stylos rouges à la tâche. Mon principal regret est de ne pas avoir réussi à te présenter l'article sur lequel il n'y aurait pas eu besoin de rouge (mais était-ce vraiment un rêve accessible ?). J'ai cependant grande confiance que toutes tes remarques m'ont marquées, et qu'elles continueront de m'influencer dans les années à venir.

Thank you, Oliver, to have accepted the duty of being one of my "rapporteur". You gave me excellent advices to improve my thesis, and our two-hour talk, which began with a coffee-break, was very pleasant.

Merci aussi à toi, Denis, pour tes remarques plus critiques sur mon manuscript, ce qui m'a permis de me rendre compte des points qu'il fallait mettre plus en avant.

Merci enfin à vous, Emmanuel et Jean-Claude, d'avoir accepté si chaleureusement d'être membres du jury.

A propos de l'ambiance de travail, je voudrais à présent remercier mes principaux co-bureau le long de ces 3 années, Raphaël et Matthieu, pour n'avoir jamais été assourdissants de paroles ou de questions, m'incitant toujours à persévérer en faisant la sourde oreille à mes questions fugaces, plutôt que de me laisser tomber dans la facilité en écoutant simplement vos réponses. Merci aussi d'avoir su rester stoïques devant mes déboires de mise à jour du système. Bien entendu, ne prenez rien au sérieux de ce qui est au dessus, à part le merci.

Merci également à tous les autres membres de l'équipe, Loris, Véronika, Emmanuel, Cédric, Anne, Clément, et tous les autres, qui donnent une ambiance magique à ces bureaux ;

Merci à Caroline, Sylvie, Corinne, et Isabelle, pour leur sourires constant, même après que je sois parti aux États-Unis sans billet d'avion (mais ce n'était pas 100% ma faute) ;

Un grand merci à Marie, pour m'avoir permis de concilier une thèse sur Lyon et une vie

conjugale sur Montpellier. Merci également pour tous ces plats que tu m'as fait découvrir, je n'ai jamais eu honte de me reservir (Miam !) ;

Merci à mes parents, pour m'avoir incité à venir découvrir le climat lyonnais ; à mes beaux-parents, pour m'avoir laissé emmener leur fille ;

Merci enfin à Ln, pour avoir donné un sens nouveau à la vie après le boulot, et tant d'autres choses encore. Et une petite pensée également au Nurson que tu portes, et qui va lui aussi changer notre vie...

Contents

Chapter 1

Introduction

In 1978, a reviewer of the article of R.L. Rivest, A. Shamir, and L. Adleman "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." [120] states:

> *Granted, we are seeing the appearance of so-called microcomputers, such as the recently announced Apple II, but their limitations are so great that neither they nor their descendants will have the power necessary to communicate through a network.*[1]

Nowadays, not only microcomputers are communicating through the network, but they are intensively used on grids composed of workstations located all around the world. These grids represent a cheaper alternative to super-computers, which are composed of a reduced number of identical powerful processors. As you may expect, the heterogeneity of such grids renders scheduling problems much more difficult. Furthermore, one can understand the importance of the communications between the machines on such platforms. If you are working in Lyon and are planning to enroll more computers to execute some application, you want to know the differences between using an additional computer from Knoxville or one from Grenoble, and so you need to know how long it is going to take to send a message from one computer to the other, and what may be the interferences.

That is why one needs to use realistic communication models when working on these grids, and that is one of the reasons I mainly focused during these last three years on communication-aware scheduling.

In this thesis, I will look at the scheduling of different applications on a master-worker platform — which is one model of distributed heterogeneous platforms. The common point of my work is to always use a realistic model to take into account the communications between the machines of the platform.

Models

In Chapter 2, I will discuss platform models, applications, scheduling techniques, and also envision the objectives that we may consider. I will introduce the master-worker platform, and justify its use (Section 2.2). I will also detail the models that we have at our disposal to represent computations (Section 2.3) and communication (Section 2.4) in large-scale architectures. Each time, I will mainly focus on the models that we will use in the following chapters. Then I will present two kind of applications, divisible load and bag-of-tasks applications (Section 2.5). I will

[1] published in [124]

point out the importance of scheduling identical independent tasks from the scheduler point of view, and I will justify this hypothesis by giving real-life applications that involve such tasks. Next, I will initiate a discussion about scheduling models (Section 2.6), i.e., on the information related to the platform and to the applications that we suppose we have at our disposal when trying to find a good schedule. Of course, the more information you have, the better algorithm you can find, but it is not always realistic to assume that everything is known beforehand. I will present the three classes of scheduling models that we will see in detail in the next chapters. Finally, I will give an overview of different objective functions that we will try to minimize, and I will justify their importance (Section 2.7). The last section (Section 2.8) will introduce some notations in order to quickly understand which model we use, and what is the problem we are working on.

The difficulty of communication-aware scheduling

Scheduling problems are already difficult on traditional parallel machines, and homogeneous platforms. They become extremely challenging on heterogeneous clusters, where computation speeds and communication links are heterogeneous, where the network may suffer from contention problems, and where availability and performance of the machines are sometimes unpredictable. This holds true even when embarrassingly parallel applications are considered.

In Chapter 3, I will try to underline the difficulty of scheduling identical independent tasks on heterogeneous platforms while minimizing the total execution time. I will distinguish four cases: (i) the platform is homogeneous (Section 3.2.1); (ii) the heterogeneity only comes from the computations (Sections 3.2.2, 3.2.3, and 3.3.1); (ii) the heterogeneity only comes from the communication links (Sections 3.2.3, and 3.3.2); or whether the platform is fully heterogeneous (Sections 3.2.3, and 3.3.3). Not surprisingly, when both sources of heterogeneity add up, the complexity goes beyond the worst scenario with a single source.

This work has been published in [B1, B2, A2].

Matrix product

Despite the results of the previous chapter, or maybe "thanks to" them, we aim in Chapter 4 at optimizing the execution time of matrix multiplication. Thanks to the previous chapter, we know that this is a difficult problem, but this is in fact good news! Because we know that it will be challenging to do so, and we may find some improvements.

Once again, we will find that minimizing the execution time is a difficult problem (Section 4.3). That is why we will stop trying to minimize it. If we forget that is our objective, what is left ? We want to perform a matrix multiplication on a grid, which means that the machines can be far from each other. And as we focus on introducing realistic communication models, the communication of elements of the matrices between the machines may be the bottleneck of the application. Thus, we concentrate on minimizing the communication volume.

First we will proceed with the analysis of the total communication volume that is needed in the presence of memory constraints (Section 4.4), and we develop an algorithm whose aim is to minimize these communications (Section 4.4.2). In order to apprehend the solution for heterogeneous platforms, we first deal with homogeneous platforms (Section 4.5), and we propose a scheduling algorithm that includes resource selection. Then we address the design of algorithms for heterogeneous platforms (Section 4.6), which turns out to be a truly challenging algorithmic task. And we briefly discuss how to extend previous approaches to LU factorization (Section 4.7)

before reporting several MPI experiments (Section 4.8).

This work has been published in [B3, A1, B4].

Steady-state

Even if the problem of minimizing the makespan is proved to be hard for heterogeneous platform, the last chapter showed that there is still some hope. In Chapter 5 we present a new way to bypass the difficulty.

Indeed, because of the time of deployment and of the difficulty of implementation on grid platforms, one can assume that grid users will deploy larger applications, which are more cost-effective. In this chapter we look at the case where applications composed of a huge number of tasks are submitted to the platform.

For such applications and platforms, I will find a better objective than the minimization of the makespan. In particular, when focusing not on the makespan but on steady-state (which occurs necessarily when the number of tasks increases), problems become much easier to solve and it is possible to find a schedule that is asymptotically optimal. The steady-state is computed thanks to linear programming, which enables to cope with the heterogeneity of computations and communications (Section 5.3), with the affinity of tasks with certain types of machines (Section 5.4), and with energy consumption constraints (Section 5.5). We also give a description of the polynomial scheduling that achieves the steady-state, and we show its asymptotic optimality.

Part of this work has been published in [B5]. All the work about energy minimization (Section 5.5) is still in progress, and has not been published yet.

Chapter 2

Framework

The objective of this chapter is to give a broad survey of the objects that we deal with throughout this thesis.

We start by describing the different platforms (Section 2.1), the computation models (Section 2.3) and the communication models (Section 2.4). Then, we discuss the applications (Section 2.5) and the scheduling models (Section 2.6). Finally, we present the different objectives that we try to optimize (Section 2.7), and notations that summarize all the characteristics of a problem (Section 2.8).

2.1 Large scale platforms: from super-computers to desktop grids

Since the sixties, the first parallel machines, also called "super-computers", have been created by aggregating a large number of identical processors, connected by a powerful communication network. Some super-computers, Cray, IBM or SGI, were built and had their moment of glory. However, these machines were very expensive, and not everyone who needed computing power could afford to book some hours on them. In the nineties, the collapse of companies marketing the super-computers was caused by the development of the clusters of workstations.

Theses clusters came from the observation that is was cheaper to gather machines spread over remote existing sites. This development was possible thanks to constant effort in terms of design and software development (particularly with libraries such as ScaLAPACK [33], PVM [70] and MPI [73, 88], and the development of asynchronous programming and threads) and the presence of technological advances in public machines (high-speed network, super-scalar processors, etc...). Heterogeneous networks of workstations are now ubiquitous in university departments and companies. They represent the typical poor man's parallel computer: Running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying super-computer hours. When implementing algorithms on such platforms, the idea is to make use of all available resources, namely, slower machines in addition to more recent ones. But the heterogeneity of such platforms makes the development of efficient programs harder, and hence their use is reserved for relatively simple parallel applications, which do not require a strong synchronization between the computation units.

The platform topology that we consider consists of network links with various characteristics to clusters of heterogeneous processors, as depicted in Figure 2.1.

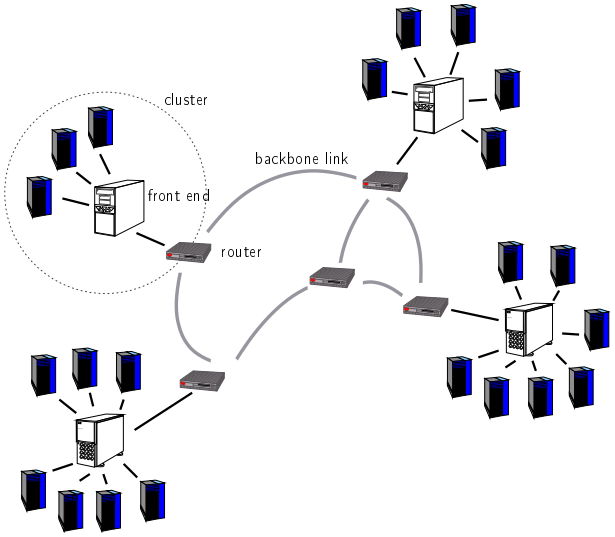


Figure 2.1: Computing platform model.

2.2 The Master-Worker platform

The master-worker platform is a centralized model to represent a grid of workers. In these platforms, one machine, the master, plays a special role in the scheduling process; it is the processor that holds the scheduling algorithm, and, most of the time, the tasks that the workers have to process. For that reason, it also has a special place in the communication network. We assume that the master has no processing capability, and is exclusively dedicated to the scheduling and to communications.

In this thesis, we supposed that the master can communicate to each worker, but the workers cannot communicate with each other. This lead to a special communication network: the *star network* (see Figure 2.2), also called single-level tree. On such a platform $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$ the master P_0 (also called P_{master}) is located at the root of the tree, and is connected to its p workers $P_1, \ldots P_p$. In practice, this star network can be a logical overlay built upon a different physical interconnection network. In a cluster where processors are connected through an over-dimensioned switch, all pair-wise communications could happen in parallel, without influencing each other, if the limiting factor is the communication capabilities of each of the processors.

This model can be applied to distributed Grid platforms, that aggregate multiple parallel computing platforms. These platforms can be easily modeled as single-level trees, where each leaf is a worker and the root is the master holding the application's input data.

One can also remark that parameter sweep applications [42] and BOINC-like computations [35] usually organize participating processors into a master/worker platform.

The main two components of the considered computing platforms are the computing resources themselves, and the interconnection networks. We will discuss about models for both in the next sections. The memory and data storage capacities are often ignored but they may also play a great role in our algorithmic problems, as we will see in Chapter 4.

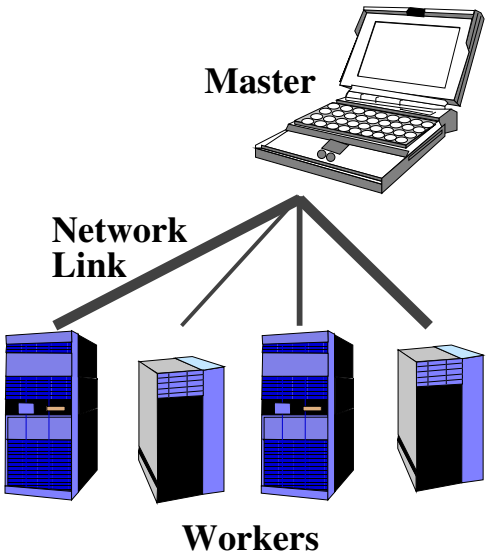


Figure 2.2: The master-worker platform.

2.3 Computation models

To design computation models, one needs to know what kind of resources we have (homogeneous, with different speeds, unable to perform some kind of tasks, ...), and how the tasks are computed on them (time-sharing, atomic computations, etc...).

2.3.1 The processors

Long time ago, the main platforms were exclusively composed of homogeneous processors. Even if nowadays, most platforms are heterogeneous, working on homogeneous platforms helps to design efficient algorithms. In this thesis, we will use such platforms as baseline references, or to stress the impact of heterogeneity on the complexity of problems or on the design of solutions. The landmark feature of the platforms that we consider is the heterogeneity of their computing resources. Such a characteristic is obvious in Grid computing platforms. Scheduling theory is proposing a classification of the heterogeneity of computing resources into three platform types:

1. The *parallel and identical machines* model refers to homogeneous platforms.

2. *Uniform machines* are platforms where the execution time of an application on a processor is equal to a constant only depending on the machine, times a constant only depending on the application. Basically, that means that once we know the relative speeds of the processors for one application, then they will have the same relative speeds for all kinds of applications.

3. The *unrelated machines* model is the general case. In particular, we have under this model the case of *restricted availabilities*, where an application cannot run on some machine (because, for instance, it needs some special libraries or operating system). We can also file under this model the case of identical processors which do not all have the same amount of available memory: depending on its memory requirements, an application execution will

or will not fit in the main memory of all the different processors, and thus will or will not have the same running time on them.

Besides the potential consequences of the different hardware characteristics, there may be software and operating system characteristics which may influence the way schedulers can deal with computing resources, and thus which influence the way resources can be used. Especially, there is the question of whether it is mandatory that, once started, a task is completed before any other task can be executed on the same processor. We will see in the next section computation models with different answers to that question.

2.3.2 Fluid computation

Under the *fluid computation* model, we assume that several tasks can be executed at the same time on a given worker, with a time-sharing mechanism. Furthermore, we assume that the computation rate for each task can be totally controlled. For example, suppose that two tasks A and B are executed on the same worker at respective rates α and β ($\alpha + \beta \leq 1$). During a time period Δt, $\alpha \cdot \Delta t$ units of work of task A and $\beta \cdot \Delta t$ units of work of task B are completed. These computation rates may be changed at any time during the computation of a task.

2.3.3 Atomic computation

Another computation model, the *atomic computation* model, assumes that only a single task can be computed on a worker at any given time, and this execution cannot be stopped before its completion (no preemption).

2.3.4 Synchronous start

Under both previous computation models, a given worker cannot start execution before it has terminated the reception of the message containing the task from the master; similarly, it cannot start sending the results back to the master before finishing the computation. However, there exists a variant to the *fluid computation* model, called *synchronous start* computation: in this model, the computation on a worker can start at the same time as the reception of the communication starts, provided that the computation rate is smaller than, or equal to, the communication rate (the communication must complete before the computation). This models the fact that, in several applications, only the first bytes of data are needed to start executing a task.

As a remark, it has been observed that initiating computations on Grid resources may incur a latency, i.e., a constant occupation time, independent of the computation size.

2.4 Communication models

Traditional scheduling models enforce the rule that computations cannot progress faster than processor speeds would allow: limitations of computation resources are well taken into account. Curiously, these models do not make similar assumptions for communications: in the literature, an arbitrary number of communications may take place at any time-step [146, 37]. In particular, a given processor can send an unlimited number of messages in parallel, and each of these messages is routed as if it was alone in the system (no sharing of resources). As surprising as it

may now be, this historical position can be easily explained because "introducing communication costs complicates matters a lot" (for instance, see Chapter 2 of [60]).

However, it can be truly difficult to define a model of communication cost on grid. Indeed, if we consider an application on a grid, we are facing several problems. First, we cannot consider that the network is dedicated to our particular application, and it is not possible to predict the external load on the network. Second, we do not know, most of the time, the network topology. And last, even when assuming an exact knowledge of the physical network topology and of the external load, the traffic simulation of the packages according to network protocols is a very long task, as shown by the work of NS [108], for which the simulation time of a data transfer was far longer than the transfer itself. Overall, such a precision is not compatible with the design of efficient scheduling algorithms.

2.4.1 Communication time

An important aspect of the model is network latency. It is well-known that a reasonable approximation of the time required to send x bytes of data over a network link is affine of the form $\alpha + \frac{x}{B}$, where α is the time required for a zero byte message to travel from the source to the destination, or *latency*, and B is the data transfer rate.

Even if some research proposed more sophisticated models as early as ten years ago, many scheduling works in the divisible load scheduling area (Section 2.5.1), whose model allows for sending tasks of arbitrary size, have assumed linear transfer times $\frac{x}{B}$. Most of these works assumed a linear transfer time in order to make it possible to derive elegant solutions to certain scheduling problems. However, ignoring latencies may lead to flawed solutions, as there is no prohibitive cost to sending large numbers of very small messages. For example, the work in [29] developed a multi-round divisible load scheduling algorithm, and noted that the linear model implies an infinite number of rounds, where an infinitesimal amount of work is sent out at each round. However, it is true that taking latencies into account adds a significant amount of complexity to the scheduling problem, and we can bypass the problem of sending infinitesimal amounts of work by considering bag-of-tasks applications (Section 2.5.2), i.e., applications where the size of each task is fixed.

Throughout this thesis, we will suppose to have a linear communication model, or we will consider the communication time of the tasks, thus taking into account latency overheads.

2.4.2 Communication behavior

The communication behavior is supposed to clarify what happens in case of several sends. Some models specify that a machine can communicate to only one other at any time, while other models allow for an arbitrary large number of communications to take place at any time-step.

Macro-data-flow

One of the first model was the *macro-data-flow* model (for instance see the survey papers [51, 63, 109, 130] and the references therein). This model takes into account communication delays as follows: let task T be a predecessor of task T' in the task graph; if both tasks are assigned to the same processor, no communication overhead is incurred and the execution of T' can start immediately at the end of the execution of T; on the contrary, if T and T' are assigned to two different processors, the computation of T' cannot start earlier than the completion time of T plus a communication delay which is a function of the two tasks and the two processors involved.

The major flaw of this macro-data-flow model is that communication resources are not limited in this model. The first problem is that a processor can send (or receive) any number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model) and an unlimited total communication bandwidth. The second problem is that the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously occur on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic in general case. These flaws are far from being only theoretical: the execution times predicted using such a model can be unrelated to the actual execution times [133, 95].

One-port model

The more restricted model is the *one-port* model [30, 31, 134, 132]. Combined with a master-worker platform, this model implies that the master can only send data to, and receive data from, a single worker at a given time-step, so that the sending operations have to be serialized. Suppose for example that the master has a message of size x to send to worker P_u. If the transfer starts at time t, then the master cannot start another sending operation before time $t+\alpha+x/B$ under affine communication times model, where α is the latency and B the bandwidth, or before $t + x/B$ under linear model.

This *one-port* model naturally comes in two flavors with return messages, depending upon whether we allow the master to simultaneously send and receive messages or not. In the unidirectional variant, a processor cannot be involved in more than one communication at a given time-step, either a send or a receive. In the bidirectional model, a processor can send and receive in parallel, but at most from a given neighbor in each direction. In both variants, if P_u sends a message to P_v, both P_u and P_v are blocked throughout the communication. If we do allow for simultaneous sends and receives, we have the *two-port* model. Usually, a processor is supposed to be able to perform one send and one receive operation at the same time. But in this thesis we mostly concentrate on the true *one-port* model, where the master cannot be enrolled in more than one communication at any time-step.

This model is more *realistic* than the standard models from the literature, where the number of simultaneous messages involving a processor is not bounded. Bhat, Raghavendra, and Prasanna [30, 31] advocate the use of the bidirectional one-port model because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network". Experimental evidence of this fact has recently been reported by Saif and Parashar [123], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a hundred kilobytes. Their result hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2.

The one-port model also fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi, Moorthy, and Panda [7], Liu [103], and Khuller and Kim [89]. In this simpler model, the communication time only depends on the sender, not on the receiver. In other words, the communication speed from a processor to all its neighbors is the same. This would restrict the study to bus platforms instead of general star platforms.

Finally, we note that some papers [8, 11] depart from the one-port model as they allow a

sending processor to initiate another communication while a previous one is still on-going on the network. However, such models insist that there is an overhead time to pay before being engaged in another operation, so there are not allowing for fully simultaneous communications.

Bounded multi-port

Some time after Saif and Parashar's study [123], recent multi-threaded communication libraries such as MPICH2 [73, 88] now allow for initiating multiple concurrent send and receive operations, thereby providing practical realizations of the multiport model.

Under the *bounded multiport* communication model [79], the master can send/receive data to/from all workers at a given time-step. However, there is a limit on the amount of data that the master can send per time-unit, denoted as BW. In other words, the total amount of data sent by the master to all workers each time-unit cannot exceed BW. Intuitively, the bound BW corresponds to the bandwidth capacity of the master's network card; the flow of data out of the card can be either directed to a single link or split among several links indifferently, hence the multiport hypothesis. The bounded multiport model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. Simultaneous sends and receives are allowed (all links are assumed bi-directional, or full-duplex).

Interferences

If a processor is communicating while it is computing, the computations may have a significant and negative impact on the available bandwidth [91, 92]. Also, when several communications happen simultaneously on the same physical link, the fact is that they may impact the performance of each other. In other words, they may have to "share" the available bandwidths. And "how" depends on the nature of the link [44].

However, in the rest of this thesis, we will suppose that no interference occurs during the communications, because of the communications model (one-port), or of the network topology. We also always assume that computations can be overlapped by independent communications, without any interference.

2.5 Applications

In the introduction, we underlined the fact that scheduling on heterogeneous platform was difficult. That is why we want to take advantage of the regularity of applications which are executed on these platforms. Indeed, on unstable distributed platforms, the execution of a parallel application requires complex checking that no failure occurs, and that no message gets lost, in order to restart the tasks that have failed. That explains why the majority of applications using these grids are simple and naturally robust. A typical application consists of a large number of independent tasks: if a job is interrupted, or the transmission of the result fails, one can just repeat this task later, on another processor, without any cost for the rest of the application.

Obviously, there does not exist an application model which would perfectly describe every application one could ever encounter. On the contrary, there exist many different models, most of them being specific to a certain type of applications. We focus here on two application models of independent tasks. Of course there exist lots of other models.

2.5.1 Divisible Load applications

Divisible load applications consist of an amount of computation, or load, that can be divided into any number of independent pieces, or *parts*. We can consider that the processing time of each part is small compared to the total time to process the whole input. In that case, the total input can be divided into *chunks* of arbitrary sizes, which may be processed in any order, and each chunk can itself contain an arbitrary number of small parts. This corresponds to a perfectly parallel job, because the application can be decomposed into sub-tasks, where every sub-task can itself be processed in parallel, on any number of workers and in any order. Such applications are amenable to the simple master-worker programming model, and can therefore be easily implemented and deployed.

The divisible load model is a good approximation for applications that consist of a large number of identical, low-granularity computations, and has thus been applied to many real-world scientific applications. For examples, pattern searching applications in computational biology [76], video compression applications [115], volume rendering applications that are used for scientific computing and biomedicine [49, 143] and even Data mining applications [136] fit the divisible load model. These applications have different characteristics, in term of total running time, total data size, and computation-communication ratio. Experiments have been conducted that highlight and quantify the diversity in application characteristics and we refer the reader to [141, 19] for more details.

2.5.2 Bag-of-Tasks applications

Another application model is the bag-of-tasks application model, where we suppose that an application is made of a given number of independent sub-tasks, and that each of these sub-tasks is atomic. It can be seen as the discrete version of the divisible load model. Therefore, in this model the size of the sub-tasks is fixed, contrarily to the divisible load model where the scheduler can freely define the size of any sub-task.

Their study is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [126], factoring large numbers [57], the Mersenne prime search [117], and those distributed computing problems organized by companies such as Entropia [64]. Bag-of-tasks are well suited for computational grids, because communication can easily become a bottleneck for tightly-coupled parallel applications.

Condor [102] and APST [27, 42] are among the first projects providing specific support for such applications. Condor was initially conceived for campus-wide networks [102], but has been extended to run on grids [67]. While APST is user-centric and does not handle multiple-applications, Condor is system-centric. Those two projects are designed for standard grids but more recent and active projects like OurGrid [54] or BOINC [35] target more distributed architectures like desktop grids. BOINC [35] is a centralized scheduler that distributes tasks for participating applications, such as SETI@home, ClimatePrediction.NET, and Einstein@Home. The set of resources is thus very large while the set of applications is small and very controlled. OurGrid is a Brazilian project that encourages people to donate their computing resources while maintaining the symmetry between consumers and providers. All these projects generally focus on designing and providing a working infrastructure, and they do not provide any analysis of scheduling techniques suited to such environments.

Identical independent tasks

The *divisible load* and *bag-of-tasks* models correspond both to trivially parallel applications, that is, to applications whose sub-tasks can be executed in any order or even concurrently. There exists a more general scenario, where one of the application's sub-tasks may produce an intermediate result which will later be used as an input data by another sub-task (we then say that the latter sub-task depends on the former).

But in this thesis, we will focus on the scheduling of independent tasks, and these tasks will be of identical size. The hypothesis of having same-size tasks, in terms of communications (volume of data sent by the master to the slave which the task is assigned to) and of computations (number of flops required for the execution), is a very important one if we expect to find efficient algorithms. Indeed, minimizing the total execution time of different-size tasks on a homogeneous platform reduced to a master and two identical slaves, without paying any cost for the communications from the master, and supposing all tasks are available to the master at the beginning, is already an NP-hard problem [1] [68]. In other words, the simplest version is NP-hard on the simplest (two-slave) homogeneous platform.

However, working under such model is still relevant, as we point out that many important scheduling problems involve large collections of identical tasks [43, 1]. Furthermore, this kind of framework is typical of a large class of problems, including parameter sweep applications [42] and BOINC-like computations [35].

Release dates and due dates

In the contexts of scheduling multiple applications, it is reasonable to assume that all applications will not be available at the same time, but on the contrary that they will arrive over time. In such a case, each application will have its own *arrival date*, or *release date*, before which the scheduler will not be able to schedule any task of the application.

Besides release dates, a task may also have a *due date*, before which it should be processed or a penalty should be paid. Or it may have a hard *deadline* before which it must be completed. Then it will be of the scheduler responsibility to take into account all these constraints.

2.6 Scheduling

In this section, we present the three classes of scheduling models that we will see in detail in the next chapters. They detail the application-related information that we have (computation size, number of tasks, release dates, etc...) when trying to find a good schedule.

We will always suppose that we have a clairvoyant model about the platform, which means that we know the computation speed of each processor and the bandwidth of each communication link. Such a model discards the problem of instability from sharing the resources with other users, and supposes that we have the computing platform dedicated to our own usage for the time of experiments.

2.6.1 Beforehand

In the simplest model, we know beforehand all the application characteristics and all the applications to be scheduled are available to the master (that means that the release dates of all

[1] reduction from 2-partition

applications are the same). This model will be used in Chapter 4 in the context of matrix multiplication. Such assumption is realistic here, because we will focus on the minimization of a single matrix product.

2.6.2 Offline

If all release dates and characteristics of the applications are known from the start, the model is said to be *offline*; if application characteristics are not known by the scheduler before their release dates (and theses release dates are unknown as well), we have an *online* model.

The offline case is mainly theoretical and is used as a baseline to study the complexity of the scheduling problems and to assess the quality of online solutions. However, it can be used on applications such as the decoding of a video stream, which is a rare example where we can know beforehand each of the many release dates.

2.6.3 Online

The online scheduling is only relevant for problems with release dates, because mainly the release times and sizes of incoming tasks are not known in advance. This model is interesting as it brings uncertainties. Such dynamic scheduling problems are more difficult to deal with than their static counterparts, the offline problems (for which all task characteristics are available before the execution begins) but they encompass a broader spectrum of applications.

Variants

In-between we may have a semi-clairvoyant model, where one only knows partial information about the future, like for example the total number of tasks, but not their release dates. Such model will be used in Chapter 3.

2.7 Objectives

The very first question of a scheduling problem is: what is our *objective*? what do we want to optimize? Here are presented many classical objective functions that we will work with in this thesis.

2.7.1 Metrics based on completion times

The most common objective function in the (parallel) scheduling literature is the *makespan*: the maximum of the task termination times. If all tasks are part of a same job, the makespan is the total job execution time. Makespan minimization is conceptually a system-centric approach, seeking to ensure efficient platform utilization. Makespan minimization is meaningful when there is only one user, when all jobs are submitted simultaneously and have the same importance. If jobs are independent or belong to several users, one may consider the *average completion time*, also called *sum completion time* or *total completion time*.

When the number of independent tasks is large, we can relax the problem of minimizing the total execution time into maximizing the throughput, i.e., the average (fractional) number of tasks executed per time-unit. This objective function is used in Chapter 5, and will be proved very useful in order to find efficient algorithms.

2.7.2 Metrics based on flow times

When jobs are arriving over time, the makespan metric is meaningless. Indeed, consider a parallel platform where at some time a bunch of jobs arrive simultaneously, and then a last job arrives at a time when all previous jobs would have completed if all were scheduled on the slowest processor. Then, as long as the schedule optimizes the execution of the last job, it has an optimal makespan whatever its processing of the initial bunch of jobs. When jobs arrive over time, the metric must take their release dates into account. The relevant parameter is no longer the date at which all jobs are completed, but the time they spend in the system, from their release dates to their completion times. This duration is called the job *flow time* or *response time*. This flow time is equal to the time the job waited before starting being processed (the so-called *wait-time*) plus its processing time. Flow-time based metrics are job-centric metrics. One can then either optimize the *maximum flow* time or the *average flow* time (also called *total flow* time or *sum-flow* time).

These flow-based metrics tend to favor long jobs to the detriment of short ones, as long jobs are more constraining. To overcome this problem, one approach is to look at the *stretch* [24] or *slowdown* [66]. The stretch of a task is the ratio of its flow time over its size, and so can then be seen as the slowdown it experiences when the system is loaded. To take into account the affinity of some tasks with some particular machines (e.g., unrelated machines of Section 2.3.1), we adapt the stretch definition to deal with the unrelated machines context. We thus define the slowdown as the runtime experienced on the loaded system divided by the runtime that would have been achieved if the system had been dedicated (Chapter 5.4).

2.7.3 Max-based vs. sum-based metrics

Intuitively, algorithms targeting max-based metrics ensure that no job is *left behind*. Such an algorithm is thus extremely "fair" in the sense that everybody's cost is made as close to the other ones as possible. Sum-based metrics tend to optimize instead the *utilization* of the platform. Unfortunately, sum-flow and max-flow cannot be optimized simultaneously (to obtain non-trivial competitive ratios for online schedulers). This is also the case of the sum-stretch and max-stretch metrics. As a consequence, it should be noted that any competitive algorithm optimizing the sum-flow metric and having a non trivial competitive ratio has the particularly undesirable property of potential starvation, i.e., that some jobs may be delayed to an unbounded extent. By contrast, max-flow time minimization cannot suffer from this problem as starvation induces indefinitely increasing (max-)flow. Furthermore, the starvation problem identified for sum-flow minimization is inherent to all sum-based objectives. In a system where fairness matters, sum-based metrics cannot be used.

2.7.4 Energy consumption

Energy consumption is obviously an important characteristic for battery-powered systems such as laptops and sensors. For more classical computing systems, reducing the energy consumption may also be important either because of heat-dissipation problems [139] or just to decrease the huge energy bills of supercomputers [50]. Energy consumption is seldom used as the only objective; one usually attempts to reach a trade-off between energy consumption and some other metrics like makespan. Energy consumption may be taken into account a priori when building the schedule (for instance by using mechanisms such as *dynamic voltage scaling* [39]), or a posteriori to assess which scheduling policy is the most energy-efficient [50].

In the second part of Chapter 5, we will try to incorporate the energy minimization to our steady-state problem.

2.8 Notations

If we want to quickly summarize all the characteristics of a problem, from the platform model up to the objective, we need to introduce some notations. Here are the notations for the platform models:

BMP-FC-SS (Bounded Multiport with Fluid Computation and Synchronous Start). This is the uttermost simple model: communication and computation start at the same time, communication and computation rates can vary over time within the limits of link and processor capabilities. We include this model in our study because it provides a good and intuitive framework to understand the results presented in Chapter 5. This model also provides an upper bound on the achievable performance, which we use as a reference for other models.

BMP-FC (Bounded Multiport with Fluid Computation). This model is a step closer to reality, as it allows computation and communication rates to vary over time, but it imposes that a task input data is completely received before its execution can start.

BMP-AC (Bounded Multiport with Atomic Computation). In this model, two tasks cannot be computed concurrently on a worker. This model takes into account the fact that controlling precisely the computing rate of two concurrent applications is practically challenging, and that it is sometimes impossible to run simultaneously two applications because of memory constraints.

OP-AC (One-Port Model with Atomic Computation). This is the same model as the BMP-AC, but with one-port communication constraint on the master. It represents systems where concurrent sends are not allowed.

There is a hierarchy among all the multiport models: intuitively, in terms of hardness,

$$\text{BMP-FC-SS} < \text{BMP-FC} < \text{BMP-AC}$$

Formally, a valid schedule for BMP-AC is valid for BMP-FC and a valid schedule for BMP-FC is valid for BMP-FC-SS. This is why studying BMP-FC-SS is useful for deriving upper bounds for all other models.

We will mainly use the realistic model **OP-AC** in Chapter 3 and 4, and discuss about all models in Chapter 5.

Unfortunately, the other notations used in this thesis to describe the variables are not uniform from one chapter to another, because of our choice of model, and to be consistent with the literature. For example, the release dates of an application k will be denoted by r_k (Chapter 3) or $r^{(k)}$ (Chapter 5) depending whether we deal with independent tasks or bag-of-tasks.

However, we can describe all our scheduling problems using the notation [98, 34] $\alpha \mid \beta \mid \gamma$:

- α denotes the platform, and the different type of processors (section 2.3.1). We use 1 when we are working on a single processor, P for platforms with identical processors, Q for platforms with different-speed but uniform processors and R for unrelated processors. We add MS to this field to indicate that we work on master-worker platforms, and note $Q_{>1}$ for platforms with at least two different-speed but uniform processors.
- β denotes all the platform and applications constraints. This is where we indicate if we have a communication-homogeneous platform or a computation-homogeneous platform. We can also precise whether or not we have tasks with release dates or deadlines. We write *online* for online problems. If this field does not contain a constraint, for example the presence of release dates, then the problem is supposed to be without release dates.
- γ denotes the objective. Along this thesis, we deal with the following objective functions:
  - the *makespan* or total execution time;
  - the maximum flow (*max-flow*);
  - the *sum-flow*;
  - the maximum stretch;
  - the average stretch.

Please refer to Appendix F at any moment to remember the notations used in each chapter.

Chapter 3

The difficulty of scheduling with communications

In this chapter, we will first start by studying a very simple problem. This problem is to compute n independent identical tasks that will arrive at different time on a master-worker platform composed of m workers $P_1, P_2, \ldots, P_m$, under the one-port model with atomic computation. We target here both offline and online problems, with several objective functions (makespan, maximum flow time, sum completion time).

In the literature, the hypothesis of independent identical tasks renders such a problem really easy to solve, as each task has the same importance. However, we will show during this chapter that, depending on the kind of platform (fully homogeneous, communication-homogeneous, computation-homogeneous, or fully heterogeneous), the problem can either be solved optimally by the most greedy algorithm, or be NP-Hard.

First we will consider the simplest platform model: a fully homogeneous platform (Section 3.2.1). On such a platform, both network links and processors are identical, which means that, as we have identical tasks, the time to send any task to any worker will be the same, and the time to compute it on any processor will be the same. For this problem, we can prove the optimality of a simple algorithm for the three chosen objective functions.

Then we will assess the impact of heterogeneity for both online and offline scheduling for the other types of platforms:

- communication-homogeneous platforms, i.e., where communication links are identical. We assume for such platforms that workers have different speeds.
- computation-homogeneous platforms, i.e., where computation speeds are identical, the heterogeneity comes solely from the different-speed communication links.
- fully heterogeneous platforms.

For online scheduling, we establish lower bounds on the competitive ratio of any deterministic algorithm. We prove one lower bound in Section 3.2.2. As three platform types and three objective functions lead to nine theorems, all of them with similar proofs, we summarise all results in Section 3.2.3, while the remaining proofs are located in Appendix A. Section 3.2.4 is dedicated to the presentation of how we build such proofs.

For offline scheduling, we develop some heuristics for communication-homogeneous platforms (Section 3.3.1) and computation-homogeneous platforms (Section 3.3.2). Section 3.3.3

is dedicated to the proof of the NP-completeness of the offline problem on fully heterogeneous platforms.

Not surprisingly, when both sources of heterogeneity add up, the complexity goes beyond the worst scenario with a single source. In other words, for the online problem on fully heterogeneous platforms, we derive competitive ratios that are higher than the maximum of the ratios with a single source of heterogeneity, while we proved that the offline problem was NP-Hard.

The main contributions of this chapter are mostly theoretical. However, on the practical side, we have implemented several heuristics, classical ones and new ones described in this work, on a small but fully heterogeneous MPI platform (Section 3.4). Our results show the superiority of those heuristics which fully take into account the relative capacity of the communication links.

Section 3.5 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 3.6.

3.1 Framework

During this chapter, we only target sets of identical tasks. This means that we always fall under the *uniform* processors framework (cf. Section 2.3.1). In other words, the execution time of a task on a processor will only depend on the processor running it and not on the task itself. Having identical independent tasks also means that their only differences are their release dates, and of course their completion times.

The tasks are simply denoted $1, 2, \ldots, i, \ldots n$. We let r_i be the release time of task i, i.e., the time at which task i becomes available on the master. In online scheduling problems, the r_i's are not known in advance. Let c_j be the time needed by the master to send a task to P_j, and let p_j be the time needed by P_j to execute a task. Finally, we let C_i denote the end of the execution of task i under the target schedule.

We will focus on three important objective functions:

- the minimization of the *makespan* (or total execution time) $max\ C_i$;
- the minimization of the *max-flow* (difference between completion time and release time) $max\ (C_i - r_i)$;
- the minimization of the *sum-flow* $\sum(C_i - r_i)$.

As we have identical tasks, the minimization of objective functions such as the stretch is equivalent to the minimization of the flow.

3.2 Online theoretical results

This section will mainly be theoretical. We are not interested into building online algorithms here (we will do so in Section 3.3).
The objective of this part is to point out the impact of the platform's heterogeneity on the difficulty of finding a close-to-optimal algorithm for one of our three objective functions under the online model.

3.2.1 Fully homogeneous platforms

In this first section, we start with the simplest possible problem: scheduling same-size tasks on fully homogeneous platforms. In fact, this problem is simple enough to find one single optimal algorithm to minimize our three different objective functions, and also optimal for online scheduling: the *Round-Robin* algorithm.

Round-Robin is a very simple algorithm, which processes the tasks in order of their arrivals, and which assigns them in a cyclic fashion to all processors; as we have p processors, *Round-Robin* will assign task i to processor i *modulo* p and will send it as soon as possible, according to the communication constraints. Here, as the platform is fully homogeneous, *Round-Robin* works as a list-scheduling strategy: it processes tasks according to their release times, the first task processed first, and sends the first unscheduled task to the processor whose ready-time is minimum, the ready-time of a processor being the time at which it has completed the execution of all the tasks that have already been assigned to it. We point out that the complexity of the *Round-Robin* algorithm is linear in the number of tasks and does not depend upon the platform size.

It is striking that this simple strategy is optimal for many classical objective functions. Our first theorem is then:

Theorem 3.1. *The* Round-Robin *algorithm is optimal for the problems*

- $P, MS \mid online,\ r_j,\ p_j = p,\ c_j = c \mid \max\ \mathcal{C}_i$,
- $P, MS \mid online,\ r_j,\ p_j = p,\ c_j = c \mid \max\ (\mathcal{C}_i - r_i)$,
- $P, MS \mid online,\ r_j,\ p_j = p,\ c_j = c \mid \sum\ (\mathcal{C}_i - r_i)$,

Proof. To prove that the greedy algorithm *Round-Robin* is optimal for our problem, we show that there is an optimal schedule under which the execution of each task starts at the exact same date than under *Round-Robin*. To prove this, we first show two results stating that we can focus on certain particular optimal schedules.

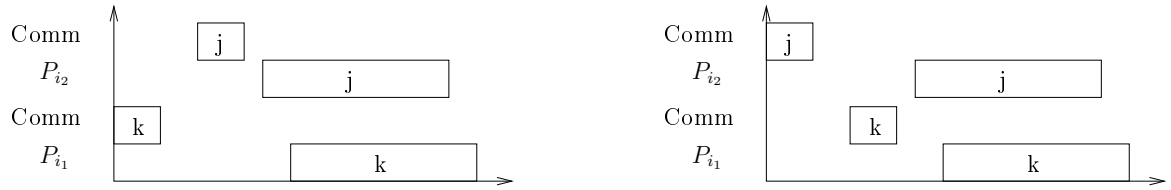
Lemma 3.1. *There is an optimal schedule such that the master sends the tasks to the workers in the order of their arrival.*

> We prove this result with permutation arguments. Let S be an optimal schedule not verifying the desired property. Remember that the master uses its communication links in a sequential fashion. Then we denote by r'_i the date at which the task i arrives on a worker. By hypothesis on S, there are two tasks, j and k, such that j arrives on the master before k, but is sent to a processor worker after k. So:
>
> $$r_j < r_k \text{ and } r'_k < r'_j.$$
>
> We then define from S a new schedule S' as follows:
>
> - If the task j was nevertheless treated earlier than the task k (i.e., if $C_j \leq C_k$), then we simply reverse the dispatch dates of tasks j and k, but do not change the processors where they are computed. This is illustrated in Figure 3.1. In this case, the remainder of the schedule is left unaffected, and the total flow remains the same (just as the makespan, and the maximum flow).

- If the task j was processed later than the task k, i.e., if $C_j > C_k$, then we send the task j to the processor that was receiving k under S, at the time task k was sent to that processor, and conversely. This is illustrated in Figure 3.2. Since the tasks j and k have the same size, the use of the processors will be the same, and the remainder of the schedule will remain unchanged. One obtains a new schedule S', having as total flow:

$$\left(\sum_{\substack{i=1 \\ i \neq j, i \neq k}}^{n} (C_i - r_i) \right) + (C_k - r_j) + (C_j - r_k) = \sum_{i=1}^{n} (C_i - r_i) \tag{3.1}$$

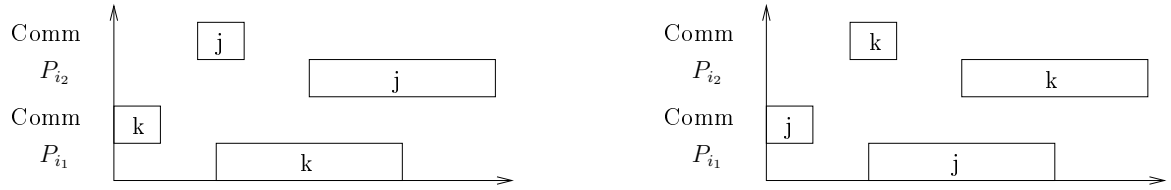
  Therefore, this is also an optimal schedule. In the same way, the makespan is unchanged and the maximum flow does not increase.



(a) Before permutation

(b) After permutation

Figure 3.1: Permutation on the optimal schedule S (case $C_j \leq C_k$).



(a) Before permutation

(b) After permutation

Figure 3.2: Permutation on the optimal schedule S (case $C_j > C_k$).

By iterating this process, we obtain an optimal schedule where the master sends the tasks according to their arrival dates, i.e., by increasing r_i's. Indeed, if one considers the set of the couples $\{(j,k) \mid r_j < r_k \text{ and } r'_j < r'_k\}$, we notice that each iteration of the process strictly increases the size of this set.

Lemma 3.2. *There is an optimal schedule such that the master sends the tasks to the workers in the order of their arrival, and such that the tasks are executed in the order of their arrival.*

We will permute tasks to build an optimal schedule satisfying this property from a schedule satisfying the property stated in Lemma 3.1. Let S be an optimal schedule in which tasks are sent by the master in the order of their arrival. From the above study, we know that such a schedule exists. Let us suppose that S does not satisfy the desired property. Then, there are two tasks j and k, such that

$$r_j \leq r_k, \quad r'_j < r'_k, \quad \text{and} \quad C_j > C_k.$$

Then we define a new schedule S' by just exchanging the processors to which the tasks j and k were sent. Then, the task j is computed under S' at the time when k was computed under S, and conversely. This way, we obtain the same total flow ($(C_j - r_k) + (C_k - r_j) = (C_j - r_j) + (C_k - r_k)$), the same makespan (since the working times of the processors remains unchanged), whereas the maximum flow does not increase.

Among the optimal schedules which respect the property stated in Lemma 3.2, we now look at the subset of the solutions computing the first task as soon as possible. Then, among this subset, we look at the solutions computing the second task as soon as possible. And so on. This way, we define from the set of all optimal schedules an optimal solution, denoted *ASAP*, which processes the tasks in the order of their arrival, and which processes each of them as soon as possible. We will now compare *ASAP* with the schedule *Round-Robin*, formally defined as follows: under *Round-Robin* the task i is sent to the processor $i \ mod \ m$ as soon as possible, while respecting the order of arrival of the tasks.

Lemma 3.3. *The computation of any task j starts at the same time under the schedules ASAP and* Round-Robin.

The proof is done by induction on the number of tasks. *Round-Robin* sends the first task as soon as possible, just as *ASAP* does. Let us suppose now that the first j tasks satisfy the property. Let us look at the behavior of *Round-Robin* on the arrival of the $(j+1)$-th task. The computation of the $(j+1)$-th task starts at time:

$$RR(j+1) = max\left\{r'_{j+1}, RR(j+1-m) + p\right\}.$$

Indeed, either the processor is available at the time the task arrives on the worker, and the task execution starts as soon as the task arrives, i.e., at time r'_{j+1}, or the processor is busy when the task arrives. In the latter case, the processor will be available when the last task it previously received (i.e., the $(j+1-m)$-th task according to the *Round-Robin* strategy) will be completed, at time $RR(j+1-m)+p$.

Therefore, if $RR(j+1) = r'_{j+1}$, *Round-Robin* remains optimal, since the task is processed as soon as it is available on a worker, and since it was sent as soon as possible. Otherwise, $RR(j+1) = RR(j+1-m)+p$. But, by induction hypothesis, we know that $\forall \lambda, 1 \leq \lambda \leq m, RR(j+1-\lambda) = ASAP(j+1-\lambda)$. Furthermore, thanks to the *Round-Robin* scheduling policy, we know that $\forall i, RR(i) \leq RR(i+1)$. Therefore:

$$\forall \lambda, 1 \leq \lambda \leq m, RR(j+1-m) \leq RR(j+1-\lambda) < RR(j+1-m) + p = RR(j+1)$$

This implies that, between $RR(j+1-m)$ and $RR(j)$, m tasks of size p were started, under *Round-Robin*, and also under *ASAP* because of the induction hypothesis. Therefore, during

that time interval, m workers were selected. Then, until the date $RR(j+1-m)+p$, all the workers are used and, thus, the task $j+1$ is launched as soon as possible by *Round-Robin*, knowing that $ASAP$ could not have launched it earlier. Therefore, $ASAP(j+1) = RR(j+1)$. We can conclude.

We have already stated that the demonstrations of Lemma 3.1 and 3.2 are valid for schedules minimizing either makespan, total flow, or maximum flow. The reasoning followed in the demonstration of Lemma 3.3 is independent from the objective function. Therefore, we demonstrated the optimality of *Round-Robin* for these three objective functions. ■

Remark (Resource selection)**.** *One can remark that if the time needed to send a task is greater than the time needed to compute it, then we only need to enroll one machine. If $c > p$, then the worker will always be able to finish its tasks before receiving new ones. Then sending tasks to one worker will be the optimal scheduling. This remark is also true for heterogeneous platforms; if the fastest communication link has this property ($c_{j_0} = min\ \{c_j\}$ and $c_{j_0} > p_{j_0}$), then sending tasks to that worker will be the optimal scheduling.*

On homogeneous platform, one can extend this to resource selection; Round-Robin does not need to enroll all workers in order to be optimal. If $m' = \lceil \frac{p}{c} \rceil$, then sending tasks to m' workers in a Round-Robin way will be optimal, as the first worker will always have finished its computation before receiving its next task.

3.2.2 Heterogeneous platforms

As we may think, online scheduling of same-size tasks on heterogeneous platforms is much more difficult. On such platforms, we show that there does not exist any optimal deterministic algorithm for on-line scheduling. This holds true for the previous three objective functions (makespan, maximum flow time, and sum of flow times).

To assess the performance of an online algorithm, one often considers the worst-case relative error between the quality of the computed solution for an instance and the quality of the corresponding optimal solution. An upper bound for the worst-case relative error is called a competitive ratio. Formally, let $f(\mathcal{A},\mathcal{I})$ denote the objective value, according to the studied optimality criterion, of the schedule produced by algorithm $\mathcal{A}$ on instance $\mathcal{I}$. Then, algorithm $\mathcal{A}$ is ρ-competitive if $f(\mathcal{A},\mathcal{I}) \leq \rho \cdot f(\mathrm{OPT},\mathcal{I})$ for any instance $\mathcal{I}$, where OPT denotes the optimal offline scheduling algorithm for our objective function.

Given a platform (say, with homogeneous communication links) and an objective function (say, makespan minimization), how can we establish a bound on the competitive ratio on the performance of any deterministic scheduling algorithm? Intuitively, the approach is the following. We assume a scheduling algorithm and we run it against a scenario elaborated by an adversary. The adversary analyzes the decisions taken by the algorithm, and reacts against them. For instance if the algorithm has scheduled a given task T on P_1 then the adversary will send two more tasks, while if the algorithm schedules T on P_3 then the adversary terminates the instance. At the end, we compute the relative performance ratio: we divide the makespan achieved by the algorithm by the makespan of the optimal solution, which we determine offline, i.e., with a complete knowledge of the problem instance (all tasks and their release dates). In one execution (task T on P_1) this performance ratio will be, say, 1.1 while in another one (task T on P_3) it will be, say, 1.2. Clearly, the minimum of the relative performance ratios over all execution scenarios is the desired bound on the competitive ratio of the algorithm: no algorithm

can do better than this bound!

Let us prove here the bound for the competitive ratio of any algorithm when minimizing the makespan on fully heterogeneous platforms.

Theorem 3.2. *There is no scheduling algorithm for the problem*

$$Q_{>2}, MS \mid online,\ r_i,\ p_j,\ c_j \mid \max\ C_i$$

whose competitive ratio ρ is strictly lower than $\frac{1+\sqrt{3}}{2}$.

Proof. Assume that there exists a deterministic online algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \frac{1+\sqrt{3}}{2} - \epsilon$, with $0 < \epsilon \leq \frac{1+\sqrt{3}}{2} - 1$. We will build a platform and an adversary to derive a contradiction. The platform is made up with three processors P_1, P_2, and P_3 such that $p_1 = \epsilon$, $p_2 = p_3 = 1 + \sqrt{3}$, $c_1 = 1 + \sqrt{3}$ and $c_2 = c_3 = 1$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes task i, either on P_1 with a makespan greater than or equal to [1] equal to $c_1 + p_1 = 1 + \sqrt{3} + \epsilon$, or on P_2 or P_3, with a makespan at least equal to $c_2 + p_2 = c_3 + p_3 = 2 + \sqrt{3}$.

At time-step 1, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 or P_3, the adversary does not send any other task. The best possible makespan is then $c_2 + p_2 = c_3 + p_3 = 2 + \sqrt{3}$. The optimal scheduling being of makespan $c_1 + p_1 = 1 + \sqrt{3} + \epsilon$, we have a competitive ratio of:

$$\rho \geq \frac{2 + \sqrt{3}}{1 + \sqrt{3} + \epsilon} > \frac{1 + \sqrt{3}}{2} - \epsilon,$$

because $\epsilon > 0$ by assumption. This contradicts the hypothesis on ρ. Thus the algorithm $\mathcal{A}$ cannot schedule task i on P_2 or P_3.

2. If $\mathcal{A}$ did not begin to send the task i at time-step 1, the adversary does not send any other task. The best makespan that can be achieved is then equal to $1 + c_1 + p_1 = 2 + \sqrt{3} + \epsilon$, which is even worse than the previous case. Consequently, the algorithm $\mathcal{A}$ does not have any other choice than to schedule task i on P_1.

Then, at time-step $\tau = 1$, the adversary sends two tasks, j and k. We consider all the scheduling possibilities:

- j and k are scheduled on P_1. Then the best achievable makespan is equal to the time needed to send three tasks to P_1 (as the j and k arrive during the communication of i), plus the time of k's computation (since i and j are computed before the reception of the next task, as $\epsilon < \frac{1+\sqrt{3}}{2}$):

$$3c_1 + p_1 = 3(1 + \sqrt{3}) + \epsilon.$$

- The first of the two jobs, j and k, to be scheduled is scheduled on P_2 (or P_3) and the other one on P_1. Then, the best achievable makespan is equal to the maximum of the

[1] Nothing forces $\mathcal{A}$ to send the task i as soon as possible.

completion time of j on P_2 and the completion time of k on P_1:

$$\begin{aligned}\max\{c_1 + c_2 + p_2 \ ; \ c_1 + c_2 + c_1 + p_1\} \\ = \max\{3 + 2\sqrt{3} \ ; \ 3 + 2\sqrt{3} + \epsilon\} \\ = 3 + 2\sqrt{3} + \epsilon.\end{aligned}$$

- The first of the two jobs j and k to be scheduled is scheduled on P_1 and the other one on P_2 (or P_3). Then, the best achievable makespan is:

$$\begin{aligned}\max\{2c_1 + p_1, \ 2c_1 + c_2 + p_2\} \\ = \max\{2 + 2\sqrt{3} + \epsilon \ ; \ 4 + 3\sqrt{3}\} \\ = 4 + 3\sqrt{3}.\end{aligned}$$

- One of the jobs j and k is scheduled on P_2 and the other one on P_3.

$$\begin{aligned}\max\{c_1 + p_1 \ ; \ c_1 + c_2 + p_2 \ ; \ c_1 + c_2 + c_3 + p_3\} \\ = \max\{1 + \sqrt{3} + \epsilon \ ; \ 3 + 2\sqrt{3} \ ; \ 4 + 2\sqrt{3}\} \\ = 4 + 2\sqrt{3}.\end{aligned}$$

- The case where j and k are both executed on P_2, or both on P_3, leads to an even worse makespan than the previous case. Therefore, we do not need to study it.

Therefore, the best achievable makespan for $\mathcal{A}$ is: $3 + 2\sqrt{3} + \epsilon$ (as $\epsilon < 1$). However, we could have scheduled i on P_2, j on P_3, and then k on P_1, thus achieving a makespan of:

$$\begin{aligned}\max\{c_2 + p_2 \ ; \ \max\{c_2 \ ; \ \tau\} + c_3 + p_3 \ ; \ \max\{c_2 \ ; \ \tau\} + c_3 + c_1 + p_1\} \\ = \max\{2 + \sqrt{3} \ ; \ 1 + 2 + \sqrt{3} \ ; \ 1 + 2 + \sqrt{3} + \epsilon\} \\ = 3 + \sqrt{3} + \epsilon.\end{aligned}$$

Therefore, we have a competitive ratio of:

$$\rho \geq \frac{3 + 2\sqrt{3} + \epsilon}{3 + \sqrt{3} + \epsilon} > \frac{1 + \sqrt{3}}{2} - \epsilon.$$

This contradicts the hypothesis on ρ. ■

3.2.3 Overview and summary

Because we have three platform types (communication-homogeneous, computation-homogeneous, fully heterogeneous) and three objective functions (makespan, max-flow, sum-flow), there remain eight bounds to be established for heterogeneous platforms. Table 3.1 summarizes the results, and shows the influence on the platform type on the difficulty of the problem. As expected, mixing both sources of heterogeneity (i.e., having both heterogeneous computations and communications) renders the problem the most difficult.

All the results presented in Table 3.1 are obtained using the same techniques, used in the previous section, and which will be described in the next section (section 3.2.4). It would thus

Platform type	Objective function		
	Makespan	Max-flow	Sum-flow
Homogeneous	1	1	1
Communication homogeneous	$\frac{5}{4} = 1.250$	$\frac{5-\sqrt{7}}{2} \approx 1.177$	$\frac{2+4\sqrt{2}}{7} \approx 1.093$
Computation homogeneous	$\frac{6}{5} = 1.200$	$\frac{5}{4} = 1.250$	$\frac{23}{22} \approx 1.045$
Heterogeneous	$\frac{1+\sqrt{3}}{2} \approx 1.366$	$\sqrt{2} \approx 1.414$	$\frac{\sqrt{13}-1}{2} \approx 1.302$

Table 3.1: Lower bounds on the competitive ratio of online algorithms, depending on the platform type and on the objective function.

be meaningless to detail each of the eight other proofs, but they can be found in detail in Appendix A. Below, we just list the characteristics of the platforms and jobs used to prove all the bounds of Table 3.1. As in the proof of Theorem 3.2, for a lower bound ρ on a competitive ratio, we assume that there exists an algorithm whose competitive ration is $\rho - \epsilon$ for a given positive value of ϵ. The potential jobs arrival times will be used by the adversary, depending on the algorithm's decisions; when one writes that the potential jobs arrival times include "X times τ" means that, if needed, the adversary will send X tasks at time τ.

Communication homogeneous platforms

Makespan : The platform consists of two processors of computation times $p_1 = 3$ and $p_2 = 7$, and of communication time $c = 1$. The potential job arrival times are: 0, 1, and 2.

Max-flow : The platform consists of two processors of computation times $p_1 = \frac{2+\sqrt{7}}{3}$ and $p_2 = \frac{1+2\sqrt{7}}{3}$, and of communication time $c = 1$; potential job arrival times: 0 and $\frac{4-\sqrt{7}}{3}$.

Sum-flow : The platform consists of two processors of computation times $p_1 = 2$ and $p_2 = 4\sqrt{2} - 2$, and of communication time $c = 1$. The potential job arrival times are: 0, 1, and 2.

Computation homogeneous platforms

Makespan : The platform consists of two processors of computation time $p = \max\left\{5, \frac{24}{25\epsilon}\right\}$, and communication times $c_1 = 1$ and $c_2 = \frac{p}{2}$. The potential job arrival times are: 0, and three times $\frac{p}{2}$.

Max-flow : The platform consists of two processors of computation time $p = 2 - \epsilon$, and communication times $c_1 = \epsilon$ and $c_2 = 1$. The potential job arrival times are: 0, and three times $1 - \epsilon$.

Sum-flow : The platform consists of two processors of computation time $p = 3$, and communication times $c_1 = 1$ and $c_2 = 2$. The potential job arrival times are: 0, and three times 2.

Fully heterogeneous platforms

Makespan : The platform consists of three processors of computation times $p_1 = \epsilon$ and $p_2 = p_3 = 1 + \sqrt{3}$, and of communication times $c_1 = 1 + \sqrt{3}$ and $c_2 = c_3 = 1$. The potential job arrival times are: 0, and two times 1.

Max-flow : The platform consists of three processors of computation times $p_1 = \epsilon$, and $p_2 = p_3 = 3 + 2\sqrt{2}$, and of communication times $c_1 = 2(1 + \sqrt{2})$ and $c_2 = c_3 = 1$. The potential job arrival times are: 0, and two times 2.

Sum-flow : The platform consists of three processors of computation times $p_1 = \epsilon$, and $p_2 = p_3 = \tau + c_1 - 1$, and of communication times c_1 (defined in Appendix A) and $c_2 = c_3 = 1$, where $\tau = \frac{\sqrt{52c_1^2+12c_1+1}-(6c_1+1)}{4}$. The potential job arrival times are: 0, and two times τ.

3.2.4 Creating the worst platform

Until now, we have proved some competitiveness lower bounds for online scheduling. Typically, we have shown that for any online algorithm, there exists a platform and tasks distribution on which they would have made a wrong decision. But you may wonder how do we create such platforms and scenarii ? And more important, how relevant are these platforms ? Is it possible to quickly "improve" one of those platforms and find a greater lower bound ?

Well, no! (or at least we hope so!)

In fact, these platforms were built in such a way that the worst competitive ratio could be achieved according to these scenarii. Let us have a look on the problem $Q_{>2}, MS \mid \text{online}, r_i, p_j, c_j \mid \max(C_i - r_i)$. The first thing to understand is where the competitive ratio comes from. What platform can maximize the ratio between any algorithm and the optimal schedule? Basically, the idea is to create a platform where the processor on which the optimal schedule sends the only task to be scheduled is different from the processor on which the optimal schedule sends the first task of a bunch of tasks. Here is the skeleton of our platform:

- two slow identical processors with fast communications;
- one fast processor with slow communication ($c_1 > c_2 = c_3$);
- when scheduling only one task, send it to the fastest processor ($c_1 + p_1 < c_2 + p_2 = c_3 + p_3$);
- when scheduling more than one task, do not send the first task to the fastest processor.

As we are under an online model, the scheduler that receives one task cannot distinguish if it will be the only task to be scheduled or if other tasks are coming. Of course, one scheduler can decide to wait for some time between the reception of the first task and the scheduling of it, but then it will have to pay this latency in its ratio. That is why we have to introduce another parameter which will be used by the adversary: the time τ at which the adversary will look at the algorithm's decision, and potentially send more tasks. Finally we will have to maximize the difference between the offline scheduling and the online scheduling in order to find our competitive ratio.

If we go back to our example, we can simulate our adversary. We send a first task at time 0. At time $\tau \geq c_2$ we look at the decision of any algorithm:

- either the first task has been sent to the first worker, which means a max flow of $\rho = c_1 + p_1$.
- either the first task has not been sent yet, and then the best max flow achievable is $\tau + c_1 + p_1$, which means a competitive ratio of:

$$\rho \geq \frac{\tau + c_1 + p_1}{c_1 + p_1}.$$

- either the first has been sent to one slow worker, so the max flow is greater than $c_2 + p_2$, and the ratio is:
$$\rho \geq \frac{c_2 + p_2}{c_1 + p_1}$$

The idea is to force any algorithm that wants to achieve a competitive ratio of ρ to send the first task to P_1 before τ. So the problem is to choose τ, c_1, c_2, p_1 and p_2 such as:

$$\min\left\{\frac{c_2 + p_2}{c_1 + p_1}, \frac{\tau + c_1 + p_1}{c_1 + p_1}\right\} \geq \rho.$$

Then, at time τ, the adversary will send two new tasks. We consider all possible schedulings.

- either the three tasks are on P_1 (Figure 3.4(a)), and the max flow is then:
$$\max\left\{\begin{array}{l} c_1 + p_1, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1, c_1 + 2p_1\} - \tau, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1 + \max\{c_1, p_1\}, c_1 + 3p_1\} - \tau \end{array}\right.$$

- either only the first and the last tasks are on P_1 (Figure 3.4(b)), the other one is on another processor. The max flow is:
$$\max\left\{\begin{array}{l} c_1 + p_1, \\ (\max\{c_1, \tau\} + c_2 + p_2) - \tau, \\ \max\{\max\{c_1, \tau\} + c_2 + c_1 + p_1, c_1 + 2p_1\} - \tau \end{array}\right.$$

- either only the first two tasks are on P_1 (Figure 3.4(c)), the other is on one slow processor:
$$\max\left\{\begin{array}{l} c_1 + p_1, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1, c_1 + 2p_1\} - \tau, \\ (\max\{c_1, \tau\} + c_1 + c_2 + p_2) - \tau \end{array}\right.$$

- or the two tasks are scheduled on P_2 and P_3 (Figure 3.4(d)):
$$\max\left\{\begin{array}{l} c_1 + p_1, \\ (\max\{c_1, \tau\} + c_2 + p_2) - \tau, \\ (\max\{c_1, \tau\} + c_2 + c_2 + p_2) - \tau \end{array}\right.$$

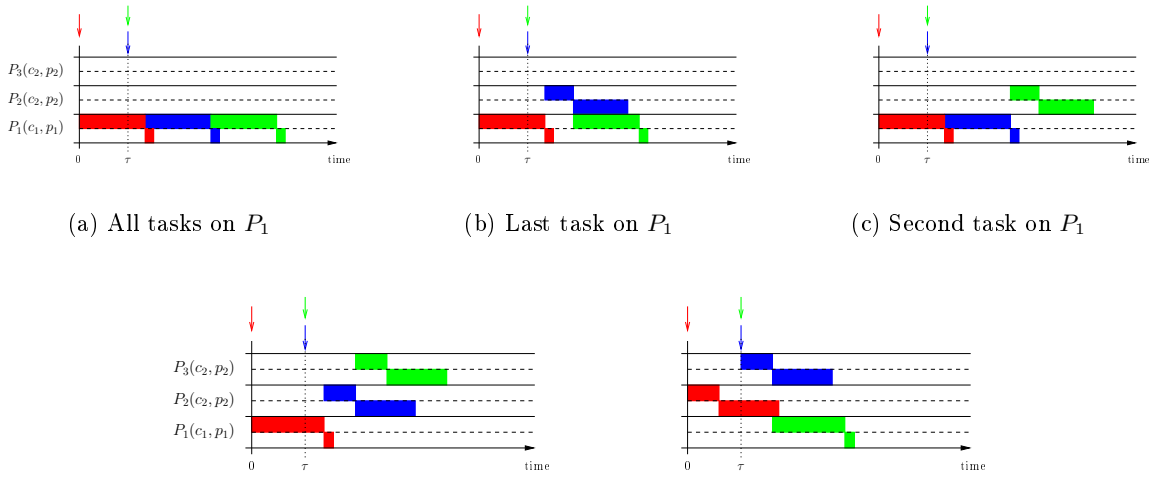
  We can simplify this max flow, as the flow of the second task will always be smaller than the flow of the third task.

- we do not need to analyze the case where two tasks are allocated on the same slow processor, because it is clearly even worse than the previous case.

We can now compare these max-flows with the offline optimal scheduling, that would have sent the first task to P_2, the second on P_3, and the third on P_1 (Figure 3.4(e)), and obtained a better max flow:

$$\max\left\{\begin{array}{l} c_2 + p_2, \\ (\max\{c_2, \tau\} + c_2 + p_2) - \tau, \\ (\max\{c_2, \tau\} + c_2 + c_1 + p_1) - \tau \end{array}\right.$$

Figure 3.3: Gantt charts for several scheduling.



(a) All tasks on P_1 (b) Last task on P_1 (c) Second task on P_1

(d) No more task on P_1 (e) Optimal solution

If we sum up, the ratio of competitiveness is defined by:

$$\rho \geq \min \left\{ \begin{array}{l} \frac{\tau+c_1+p_1}{c_1+p_1}, \\ \\ \frac{1+p_2}{c_1+p_1}, \\ \\ \min \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} c_1+p_1, \\ \max\{\max\{c_1,\tau\}+c_1+p_1, c_1+2p_1\}-\tau, \\ \max\{\max\{c_1,\tau\}+c_1+p_1+\max\{c_1,p_1\}, c_1+3p_1\}-\tau \end{array} \right. \\ \max \left\{ \begin{array}{l} c_1+p_1, \\ (\max\{c_1,\tau\}+c_2+p_2)-\tau, \\ \max\{\max\{c_1,\tau\}+c_2+c_1+p_1, c_1+2p_1\}-\tau \end{array} \right. \\ \max \left\{ \begin{array}{l} c_1+p_1, \\ \max\{\max\{c_1,\tau\}+c_1+p_1, c_1+2p_1\}-\tau, \\ (\max\{c_1,\tau\}+c_1+c_2+p_2)-\tau \end{array} \right. \\ \max \left\{ \begin{array}{l} c_1+p_1, \\ (\max\{c_1,\tau\}+c_2+p_2)-\tau, \\ (\max\{c_1,\tau\}+c_2+c_2+p_2)-\tau \end{array} \right. \end{array} \right. \\ \hline \max \left\{ \begin{array}{l} c_2+p_2, \\ (\max\{c_2,\tau\}+c_2+p_2)-\tau, \\ (\max\{c_2,\tau\}+c_2+c_1+p_1)-\tau \end{array} \right. \end{array} \right.$$

And our problem is to find τ, c_1, c_2, p_1 and p_2 which maximize this lower bound, and such as: $c_1 + p_1 < c_2 + p_2$.

In order to solve such problems, we proceed in two steps: first simplify the problem, by scaling one parameter to 1 (for example $c_2 = 1$). Then we perform a numerical resolution, in order to characterize the optimal solution (here $\tau < c_1$, $p_1 = 0$, etc). Finally, using such data,

we can simplify our system:

$$\rho \geq \min \left\{ \begin{array}{l} \frac{\tau+c_1}{c_1}, \\ \frac{1+p_2}{c_1}, \\ \min \left\{ \begin{array}{l} 3c_1 - \tau, \\ c_1 + 1 - \tau + p_2, \\ 2c_1 - \tau + 1 + p_2 \end{array} \right. \\ \frac{c_1 + 2 + p_2 - \tau}{1+p_2} \end{array} \right. = \min \left\{ \begin{array}{l} \frac{\tau+c_1}{c_1}, \\ \frac{1+p_2}{c_1}, \\ \frac{c_1+1-\tau+p_2}{1+p_2} \end{array} \right.$$

This last system can easily be maximized analytically, and we obtain the platform and τ that can be used by an adversary in order to achieve the lower bound. In this example, it get back to the platform given in section 3.2.3: $c_1 = 2(1+\sqrt{2})$, $c_2 = 1$, $p_1 = \epsilon$, $p_2 = \sqrt{2}c_1 - 1$, $\tau = 2$, $\rho = \sqrt{2}$.

3.3 Heuristics

Now that we have established lower bounds for all the online problems considered, we will study their offline counterparts, that is the situations where we know beforehand all the problem characteristics, i.e., the release dates.

3.3.1 Communication-homogeneous platforms

In this section, we have $c_j = c$ but different-speed processors. We order them so that P_1 is the fastest processor (p_1 is the smallest computing time p_i), while P_m is the slowest processor. We aim at designing an optimal algorithm for minimizing the total completion time $\max C_i$. Intuitively, to minimize the completion date of the task arriving last, it is necessary to allocate this task to the fastest processor (which will finish it the most rapidly). However, the other tasks should also be assigned so that this fastest processor will be available as soon as possible for the task arriving last. We define the greedy algorithm ***SLJF*** *(Scheduling Last Jobs First)* as follows:

Initialization– Take the last task which arrives in the system and allocate it to the fastest processor (Figure 3.5(a)).

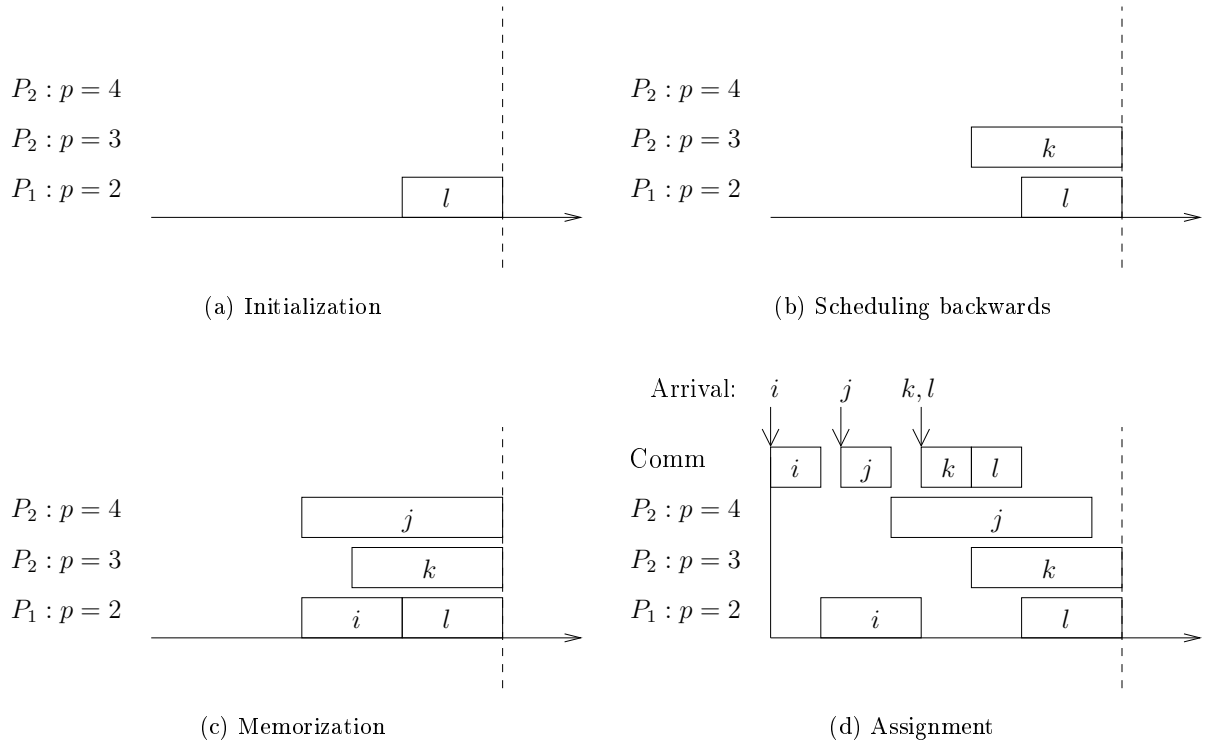
Scheduling backwards– Among the not-yet-allocated tasks, select the one which arrived latest in the system. Assign it, without taking its arrival date into account, to the processor which will begin its execution at the latest, but without exceeding the completion date of the previously scheduled task (Figure 3.5(b)).

Memorization– Once all tasks are allocated, record the assignment of the tasks to the processors (Figure 3.5(c)).

Assignment– The master sends the tasks according to their arrival dates, as soon as possible, to the processors which they have been assigned to in the previous step (Figure 3.5(d)).

This is the design of an optimal makespan minimization algorithm for the offline problem with release dates. This algorithm generalizes, and, by posing $c = 0$, provides a new proof of, a result of Simons [131].

Figure 3.4: Different steps of the ***SLJF*** algorithm, with four tasks i, j, k, and l.



Theorem 3.3. SLJF *is an optimal algorithm for the problem*

$$Q, MS \mid r_j, p_j, c_j = c \mid max\ C_i.$$

Proof. The first three phases of the SLJF algorithm are independent of the release dates, and only depend on the number of tasks which will arrive in the system. The proof proceeds in three steps. First we study the problem without communication costs, nor release dates. Next, we take release dates into account. Finally, we extend the result to the case with communications. The second step is the most difficult.

For the first step, we have to minimize the makespan during the scheduling of identical tasks with heterogeneous processors, without release dates. Without communication costs, this is a well-known load balancing, problem, which can be solved by a greedy algorithm [23]. The "scheduling backwards" phase of ***SLJF*** solves this load balancing problem optimally. Since the problem is without release dates and communication, the "assignment" phase does not increase the makespan, which thus remains optimal.

Next we add the constraints of release dates.

We denote by M the makespan achieved with release dates. Let P_j be a processor on which the last processed task is completed at the makespan (Figure 3.5). Let i be the last task executed on P_j whose processing started at its release date (in other words, we have $C_i = r_i + p_j$). Such a task exists as the processing of the first task executed on P_j began at its release date,

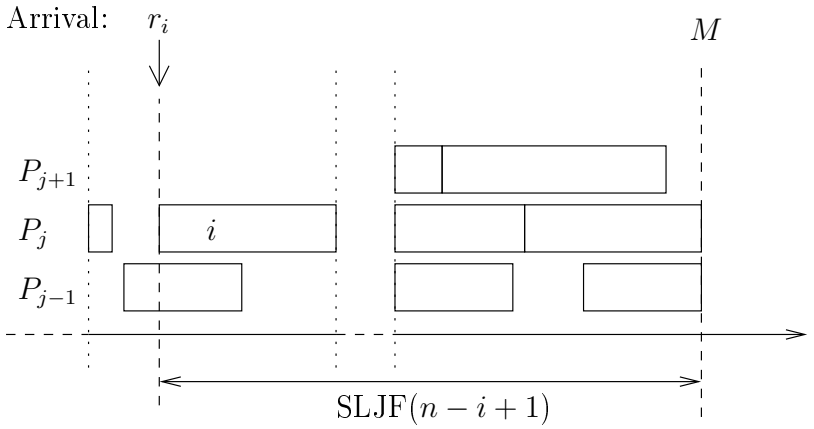


Figure 3.5: SLJF algorithm.

the scheduling, in the assignment phase, being done under the as-soon-as-possible policy. By definition of task i, processor P_j is never idle between the release date r_i and the makespan M, and is thus used $M-r_i$ time units during that interval. Our original problem is more constrained than the problem of scheduling $n-i+1$ tasks whose release dates are all equal to r_i. ($n-i+1$ corresponds to task i plus the tasks released after it.) Therefore our original problem has an optimal makespan greater than or equal to the makespan of this later and simpler problem. The simpler problem is the well-known load-balancing problem we were referring above, and we have seen that the first step of ***SLJF*** solves it optimally. Furthermore, ***SLJF*** solves it in an incremental way: the optimal solution for $k+1$ tasks is built by optimally adding a task to the optimal solution for k tasks. Therefore, ***SLJF*** optimally load-balanced the last $n-i+1$ tasks in the first step. As in this step it uses processor P_j during $M-r_i$ time units, the simpler problem of scheduling $n-i+1$ tasks whose release dates are all equal to r_i has an optimal makespan of M. From what precedes, our original problem has thus an optimal makespan greater than or equal to M, which is exactly the makespan achieved by ***SLJF***, hence its optimality.

Taking communications into account is now easy. Under the one-port model, with a uniform communication time for all tasks and processors, the optimal policy of the master consists in sending the tasks as soon as they arrive, using a *First Come First Serve* policy. Now, we consider the date at which a task is available on a worker as its *release date* for our previous problem with release dates and without communications. As a task cannot arrive sooner on any worker than this available date, and as our policy is optimal with release dates, ***SLJF*** is optimal for *makespan* minimization with release dates and homogeneous communications. ■

Online adaptation. As it only needs to know in advance the total number of tasks, but not their release dates, ***SLJF*** is also optimal to minimize the *makespan* for online problems where only the total number of tasks is known beforehand.

3.3.2 Computation-homogeneous platforms

In this section, the processors are homogeneous, i.e., $p_j = p$, but processor links have different capacities. We order the processors so that P_1 is the fastest communicating processor (c_1 is the smallest of the communication times).

We saw in section 3.2.1 that there exists an easy case where the communication links are fast enough in comparison with the computation time, and so we do not need to enroll all the workers. The same idea works here. When the time needed to send one task to all processors is greater than the time p needed by one processor to compute it, i.e., $\sum_{i=1}^{m} c_i \leq p$, we can easily prove that a variant of the scheduling policy *Round-Robin*, which sends by non-increasing communication time and sends the last task on the fastest communication link, is optimal for offline makespan minimization with release dates, as well as for online makespan minimization when the total number of tasks is known beforehand. This proof is very similar to the one of ***SLJF***. The idea is that as you have enough time to send one task to each worker, there is no resource selection to do. And as you send the last task on the fastest communication link, the makespan will be minimized.

In the general case, as we assume a one-port model, not all workers will be enrolled in the computation. Intuitively, the idea is to use the fastest m' links, where m' is computed so that the time p to execute a task lies between the time necessary to send a task on each of the fastest $m'-1$ links and the time necessary to send a task on each of the fastest m' links. Formally,

$$\sum_{i=1}^{m'-1} c_i < p \leq \sum_{i=1}^{m'} c_i$$

With only m' links selected in the platform, we aim at deriving an algorithm similar to *Round-Robin*. But we did not succeed in proving the optimality of our approach. Hence the algorithm below should rather be seen as a heuristic.

The difficulty lies in deciding when to use the m'-th processor. In addition to be the one having the slowest communication link, its use can cause a moment of inactivity on another processor, since $\sum_{i=1}^{m'-1} c_i + c_{m'} \geq p$. Our greedy algorithm will simply compare the performance of two strategies, the one sending tasks only to the $m'-1$ first processors, and the other using the m'-th processor "at the best possible moment".

Let **RRA** be a *Round-Robin* algorithm, sending the tasks on the $m'-1$ fastest links, starting with the fastest link, and scheduling the tasks in the reverse order of release times, from the last one to the first one. In other words, **RRA** sends the last task on the fastest link, the previous task on the second fastest link, and so on. Let **RRB** be the algorithm sending the last task to processor $P_{m'}$, then following the **RRA** policy. We see that **RRA** seeks to continuously use the processors, even though the communication links may sometimes be idle and processor $P_{m'}$ is always idle. On the other hand, **RRB** tries to reduce the idle time of the communication links by using one more processor at the beginning.

The intuition underlying our algorithm is simple. We know that if we only had the $m'-1$ processors with the fastest links, then **RRA** would be optimal to minimize the makespan. However, the time necessary for sending a task to each of the first $m'-1$ processors is lower than p. This means that sending the tasks takes less time than their execution.

This advance, which accumulates over tasks, can become sufficiently large to allow the sending of a task to the m'-th processor, for "free", i.e., without delaying the treatment of the following tasks on the other processors.

For a problem without release dates, we can find a closed formula to compute the makespan of each algorithm and so determine the number i of sending to processor $P_{m'}$ that will be overlapped by computations. Therefore, there only remains a small number of tasks for which the comparison between **RRA** and **RRB** is needed. Thus, our algorithm can be decomposed into two steps:

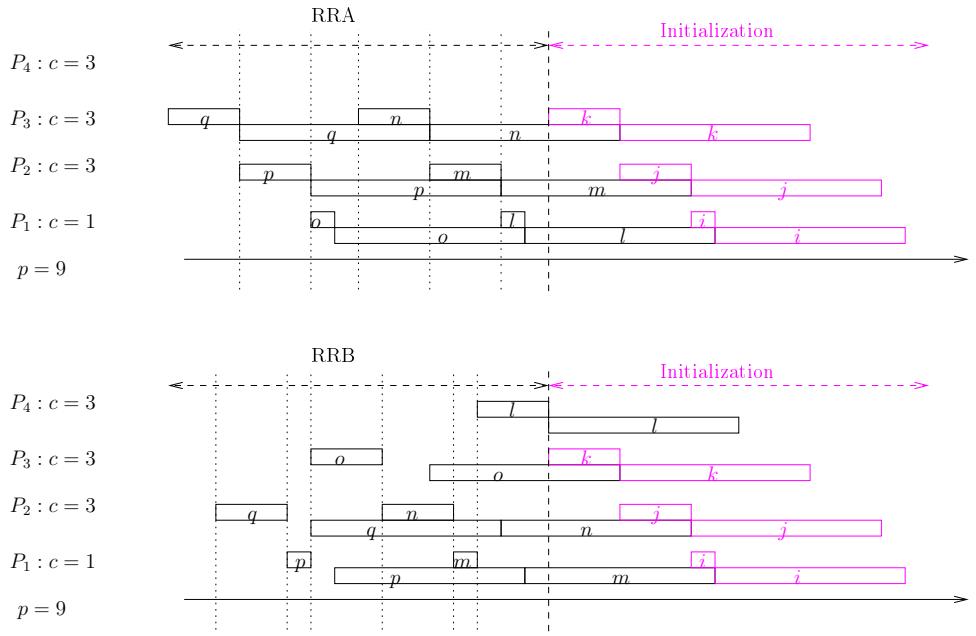


Figure 3.6: Algorithms **RRA** and **RRB**.

- determine i and k, such that $k(m'-1)+i \leq n$, i being the maximum number of tasks that can be sent to m' whose communications can be overlapped by k tasks being sent to every $m'-1$ first processors according to a **RRA** policy,
- determine the best policy among **RRA** and **RRB** to schedule the remaining $n-(k(m'-1)+i)$ tasks.

With release dates, the problem becomes of course more complicated, as the advance accumulated over tasks can be reduced because of release dates. We have to update our algorithm to take the release dates into account. We can compute the communication occupation under **RRA** in presence of release dates. As we do not know the distribution of the release dates, we have to look at the maximum of the release date of the i-th task plus the communication times to the remaining tasks to the first $(m'-1)$ processors in a cyclic way. Thus, one can know the time needed to send n' tasks according to **RRA** policy, taking into account release dates, and with the constraint that the last task is sent to P_1:

$$\mathbf{RRA}_{\text{comm}}(n') = \max_{i=1}^{n'} \left\{ r_i + \sum_{k=0}^{n'-i} c_{(k[m'-1])+1} \right\}.$$

The completion time of **RRA** with n tasks is the completion time of the last task on P_1. We know that P_1 receives during **RRA** $\left\lfloor \frac{n}{m'-1} \right\rfloor$ tasks, so the ready-time of P_1 is the maximum between the time needed to send one of these tasks to P_1, plus the completion time of all following tasks:

$$\mathbf{RRA}(n) = \max_{k=0}^{\left\lfloor \frac{n}{m'-1} \right\rfloor} \left\{ \mathbf{RRA}_{\text{comm}}(n-k(m'-1)) + (k+1)p \right\}.$$

To express the communication time of n tasks according to **RRB** is more complicated, as we have to send the tasks $n - m' + 1$ to $P_{m'}$. So the occupation of the communication is equal to the maximum of the communication time of $n - m'$ tasks under **RRA** plus the sending times to the first m' processors, and the release dates of the last tasks plus the remaining communication times:

$$\mathbf{RRB}_{\text{comm}}(n) = \max\left\{\mathbf{RRA}_{\text{comm}}(n-m') + \sum_{k=1}^{m'} c_k; \max_{i=n-m'+1}^{n}\left\{r_i + \sum_{k=1}^{n-i+1} c_k\right\}\right\}.$$

And the completion time of **RRB** is the maximum of the communication time of the last task to P_1 and the completion time of task $n - m'$ on P_1, plus computation time of the task n:

$$\mathbf{RRB}(n) = \max\left\{\mathbf{RRA}(n-m'); \mathbf{RRB}_{\text{comm}}(n)\right\} + p_1.$$

Our last parameter is the number of times we can send a task to processor m'. One can remark that **RRB** always uses a faster communication links than **RRA** to send a task, or **RRA** has to wait before using the fastest communication links. Thus, if **RRB** achieves a better completion time than **RRA** for n' tasks, then **RRB** will always have a smaller completion time than **RRA** for any greater number of tasks. So we can perform our scheduling backward, while comparing **RRA** and **RRB**. Every time that **RRB** achieves a lower completion time means that we can send another task to $P_{m'}$, and we continue the comparing of the two scheduling policies with the remaining tasks.

We call the resulting greedy heuristic ***SLJFWC*** for *(Scheduling the Last Job First With Communication)*. Algorithm 1 presents its pseudo-code version. This heuristic has complexity of $O(n + m \log m)$.

Algorithm 1: Heuristic SLJFWC for problem $Q, MS \mid r_i, p_j = p, c_j \mid C_{\max}$

```
n2 ← n − m' + 1; /* last task to consider */
n1 ← n2; /* first task to consider */
k ← 1; /* number of round */
while n1 > 1 do
    n1 ← max{(n2 − (k(m' − 1) + 1)); 1};
    Compare RRA and RRB with the tasks n1 to n2;
    if RRB best then
        n2 ← n1;
        k ← 1;
    else
        k ← k + 1;
Send according to the best scheduling the first n − (m' − 1) tasks;
Send the last m' − 1 tasks on the m' − 1 links from the slowest to the fastest;
```

3.3.3 Fully heterogeneous platform

On a fully heterogeneous platform, the problem becomes more difficult as expected. Whereas the problem of minimizing the makespan without release dates can be solved in polynomial time, the general problem with release dates can be proved to be NP-hard.

Without release dates

The problem without release dates can be solved in two ways. The first solution is due to Beaumont, Legrand, and Robert [22] who present an algorithm which, given a platform and n tasks, checks in polynomial time whether one can process the tasks on the platform in a given time T. Using this algorithm, one can find the minimum makespan by performing a binary search on T. As this binary search is over rational values, one could fear that it does not always terminate. One can however show that not only does such a binary search always terminate, but that it completes in a number of steps polynomial in the size of our problem (the proof is identical to the one that will be used for Theorem 3.5). This approach leads to a polynomial time algorithm but uses as a building block a complicated algorithm of complexity $O(n^2m^2)$.

To obtain a solution of lower complexity, we propose to reduce our problem to a problem with deadlines. Given a makespan M, we will still check whether there exists a schedule which completes all the work in time. Then, using this as a basic block, we will find the optimal makespan with a binary search. We will use *Moore's algorithm* [105] whose aim is to minimize the number of tardy tasks, i.e., which are not completed by their deadlines. This algorithm gives a solution to the $1|p_i, d_i|\sum U_j$ problem where the maximum number of tasks, among n candidate tasks, has to be processed in time on a single machine. Each task k, $1 \leq k \leq n$, has a processing time p_k and a deadline d_k before which it has to be processed.

Moore's algorithm —Algorithm 2—, is a classical algorithm in scheduling literature, and works as follows. All tasks are ordered in non-decreasing order of their deadlines. Tasks are added to the solution one by one in this order as long as their deadlines are satisfied. If a task k is out of time, the task j in the actual solution with the largest processing time p_j is deleted from the solution. Moore's algorithm runs in $O(n \log n)$.

Algorithm 2: Moore's algorithm

```
Data: a set of jobs with their deadlines and sizes: {(d_i, p_i)}_{1≤i≤n}.
Order the jobs by non-decreasing deadlines: d_1 ≤ d_2 ≤ ··· ≤ d_n;
σ ← ∅; t ← 0;
for i := 1 to n do
    σ ← σ ∪ {i};
    t ← t + p_i;
    if t > d_i then
        Find job j in σ with largest p_j value;
        σ ← σ\{j};
        t ← t − p_j;
```

In order to use Moore's algorithm, which is supposed to schedule tasks with deadlines on one machine, we will need to create tasks with deadlines. Recall we assume a one-port model, and that we only need to schedule the communications, i.e., the use of the communication link by the master (on each worker the assigned tasks are scheduled as soon as possible). Therefore, the communication link corresponds to the machine in Moore's algorithm. For the targeted makespan M, the deadlines and the computation times of the tasks will be defined as follows. We compute for each worker of our platform the deadline of each of the tasks it can receive: d_j^i denotes the deadline of the i-th last task for the worker P_j. Beginning at the makespan M, one computes when the last task has to arrive on the worker so that it can still be processed in time. The latest moment at which a task can arrive so that it can still be computed on worker P_j is

$M - p_j$. Then $d_j^1 = M - p_j$. The latest moment at which the task before the last one can arrive so that it can still be computed in time on worker P_j is $M - 2 \times p_j$. Then $d_j^2 = M - 2 \times p_j$, and so on. We denote by l_j the maximum number of tasks that worker P_j can receive: $l_j = \left\lfloor \frac{M}{p_j} \right\rfloor$. See Figure 3.7 for an example. We denote by l the total number of tasks the platform could receive: $l = \sum_{i=1}^{m} l_j$. Obviously l is an upper bound and can only be achieved if there is no communication contention, which is unlikely to happen. In other words, with these deadlines we have defined a set of l *virtual* tasks. We are going to effectively schedule as many of these virtual tasks as possible. If the number we succeed to schedule is greater than or equal to the number of tasks we actually have to schedule, n, we will have found a schedule whose makespan is no greater than M. Otherwise we will have failed.

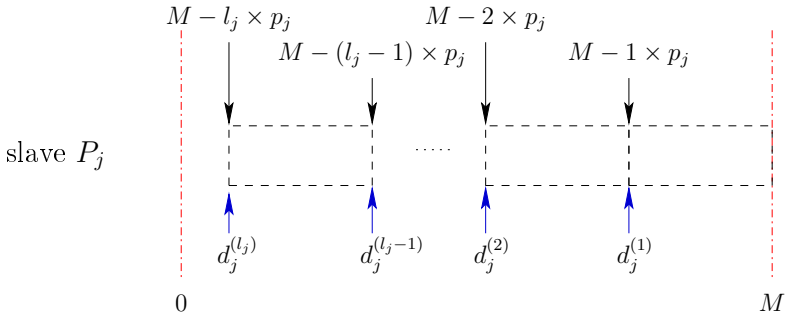


Figure 3.7: Computation of the deadlines d_j^k for worker P_j.

In our model, a virtual task j with a deadline $d_j = d_{i_j}^k$ has a processing time equal to the communication time c_{i_j} to P_{i_j}. Then the master has to decide which virtual tasks have to be sent to which workers and in which order. To solve this problem we use Moore's algorithm. Starting at time $t = 0$, the master can start scheduling the tasks on the communication link. For this purpose all the deadlines d_j are ordered by non-decreasing values. In the same manner as in Moore's algorithm, an optimal schedule σ is computed by adding one by one the tasks to the schedule: if we consider the deadline $d_j = d_{i_j}^k$, we add a task to processor P_{i_j}. So if a deadline is not met, the largest communication is suppressed from σ and we continue.

The adapted Moore's algorithm is described by Algorithm 3.

Algorithm 3: Adapted Moore's algorithm

Data: $\{(d_j, c_{i_j})\}_j$
Order the jobs by non-decreasing deadlines: $d_1 \leq d_2 \leq \cdots \leq d_n$;
$\sigma \leftarrow \emptyset$; $t \leftarrow 0$;
for $j := 1$ **to** l **do**
  $\sigma \leftarrow \sigma \cup \{(d_j, c_{i_j})\}$;
  $t \leftarrow t + c_{i_j}$;
  if $t > d_j$ **then**
    Find job k in σ with largest c_{i_k} value;
    $\sigma \leftarrow \sigma \backslash \{(d_k, c_{i_k})\}$;
    $t \leftarrow t - c_{i_k}$;
return σ;

One only needs to add a binary search to finalize our algorithm, which is described by

Algorithm 4. We bound the search of the minimal makespan with two values. As at least one machine will have to compute no less than $\lceil \frac{n}{m} \rceil$ tasks, the lower bound is the minimum time needed by one machine to compute $\lceil \frac{n}{m} \rceil$ tasks sequentially. We will suppose for simplicity of the proofs that $\frac{n}{m}$ is an integer. Depending on the communication to computation ratio, this time equals to either the time needed to make one communication with the worker and $\frac{n}{m}$ computations, or the time needed to make $\frac{n}{m}$ communications and one computation. We will call this value f_{min}, and we have:

$$f_{min} = \min_j \left\{ \frac{n}{m} \max\{c_j, p_j\} + \min\{c_j, p_j\} \right\}.$$

The upper bound, f_{max}, is the minimum time needed by one machine to compute all the tasks. We have:

$$f_{max} = \min_j \{ n \max\{c_j, p_j\} + \min\{c_j, p_j\} \}.$$

The output of Algorithm 4 is a set σ of couples of deadlines and communication times. From such a set one straightforwardly defines a schedule: the virtual tasks described by σ are sent by the master as soon as possible and by non decreasing deadlines, a virtual task (d_j, c_{i_j}) being sent to the processor P_{i_j}. This schedule has a makespan no greater than M by construction of the deadlines and by the optimality of Moore's algorithm.

Theorem 3.4. *Algorithm 4 builds an optimal schedule σ for the scheduling problem $Q, MS \mid p_j, c_j \mid C_{\max}$.*

Proof. Algorithm 4 is nothing but a binary search over the makespan M, the core of the algorithm being a call to Algorithm 3 (Adapted Moore's Algorithm) on a set of virtual tasks. Therefore, to prove the optimality of Algorithm 4 we only have to show that Algorithm 3, when called on a virtual set of tasks built for the objective makespan M, outputs a set σ containing at least n tasks if, and only if, $M \geq \mathcal{M}$, where $\mathcal{M}$ is the optimal makespan.

Let us take any value $M \geq \mathcal{M}$. From what precedes, if Algorithm 4 returns a set σ containing at least n elements, there exists a schedule whose makespan is less than, or equal to, M. Conversely, by definition of the optimal makespan $\mathcal{M}$, there exists a schedule σ^* of n tasks with a makespan less than, or equal to, M. We will prove that in this case Algorithm 3 returns a set σ containing at least n elements.

Let N_i denote the number of tasks received by P_i under σ^*. So we have $n = \sum_i N_i$. Let us denote by D the set of virtual tasks computed by our algorithm for the scheduling problem for makespan M:

$$D = \bigcup_{1 \leq i \leq m} \bigcup_{1 \leq j \leq l_i = \left\lfloor \frac{M}{p_i} \right\rfloor} \{(M - j \times p_i, c_i)\}.$$

We also define the set of virtual tasks corresponding to the N_i latest deadlines in D for each worker P_i:

$$D^* = \bigcup_{1 \leq i \leq m} \bigcup_{1 \leq j \leq N_i} \{(M - j \times p_i, c_i)\}.$$

Obviously $D^* \subseteq D$. The set of tasks in σ^* is exactly a set of tasks that respects the deadlines in D^*. The application of Moore's algorithm on the same problem returns a maximal solution. With $D^* \subseteq D$, we already know that there exists a solution with $n = |D^*|$ scheduled tasks. So Moore's algorithm will return a solution with $|\sigma| \geq |D^*| \geq n$ tasks (as there may be more than n possible deadlines in D), as we wanted. ∎

Algorithm 4: Algorithm for problem $Q, MS \mid p_j,\ c_j \mid C_{\max}$

Data: $p_i = \frac{\alpha_i}{\beta_i}, \alpha_i, \beta_i \in \mathbb{N} \times \mathbb{N}^*$, $c_i = \frac{\gamma_i}{\delta_i}, \gamma_i, \delta_i \in \mathbb{N} \times \mathbb{N}^*$
$\lambda \leftarrow \text{lcm}_{1 \leq i \leq m}\{\beta_i, \delta_i\}$;
$precision \leftarrow \frac{1}{\lambda}$;
$lo \leftarrow \min_j\{\frac{n}{m}\max\{c_j, p_j\} + \min\{c_j, p_j\}\}$;
$hi \leftarrow \min_j\{n\max\{c_j, p_j\} + \min\{c_j, p_j\}\}$;
$M \leftarrow hi$;
repeat
  for $i := 1$ **to** m **do**
    $l_i \leftarrow \left\lfloor \frac{M}{p_i} \right\rfloor$;
    for $k := 1$ **to** $\min\{l_i, n\}$ **do**
      $S \leftarrow S \cup \{(M - k \times p_i, c_i)\}$;
  if $\sum l_i < n$ **then**
    /* M is too small */;
    $lo \leftarrow M$;
  else
    $\sigma \leftarrow$ Adapted Moore's algorithm (S);
    if $|\sigma| < n$ **then**
      /* M is too small */;
      $lo \leftarrow M$;
    else
      /* M is maybe too big */;
      $hi \leftarrow M$;
      $\sigma_{\text{opt}} \leftarrow \sigma$;
  $gap \leftarrow |lo - hi|$;
  $M \leftarrow (lo + hi)/2$;
until $gap < precision$;
return σ_{opt};

Theorem 3.5. *Scheduling problem $Q, MS \mid p_j,\ c_j \mid C_{\max}$ is solvable in polynomial time by Algorithm 4.*

Proof. Thanks to Theorem 3.4, we only have to show that Algorithm 4 runs in polynomial time.

We perform a binary search for a solution in the interval $[f_{min}, f_{max}]$. As we are in heterogeneous computation conditions, we have heterogeneous p_i-values: for each $i \in [1; m]$, $p_i \in \mathbb{Q}$. The communications are also heterogeneous, so we have $c_i \in \mathbb{Q}$ for each $i \in [1; m]$. For each $i \in [1; m]$, let the representation of the values be of the following form:

$$p_i = \frac{\alpha_i}{\beta_i}, \alpha_i, \beta_i \in \mathbb{N} \times \mathbb{N}^*, \quad \text{and} \quad c_i = \frac{\gamma_i}{\delta_i}, \gamma_i, \delta_i \in \mathbb{N} \times \mathbb{N}^*,$$

where α_i and β_i are relatively prime, and also γ_i and δ_i are relatively prime.

Let λ be the least common multiple of the denominators β_i and δ_i:

$$\lambda = \text{lcm}_{1 \leq i \leq m}\{\beta_i, \delta_i\}.$$

As a consequence, for any i in $[1..m]$, $\lambda \times p_i \in \mathbb{N}$ and $\lambda \times c_i \in \mathbb{N}$. Now we have to choose the precision which allows us to stop our binary search. For this, we take a look at the possible finish

times of the workers: all of them are linear combinations of the different c_i and p_i-values. Some optimal algorithms may have some idle times, but without any loss of generality, we only look at the algorithms which send tasks and compute them as soon as possible. So if we multiply all values with λ we get integers for all values and the smallest gap between two finish times is at least 1. So the precision p, i.e., the minimal gap between two feasible finish times, is $p = \frac{1}{\lambda}$.

The maximal number of different values M we have to try can be computed as follows: we examine our algorithm in the interval $[f_{min}, f_{max}]$. The possible values have an increment of $\frac{1}{\lambda}$. So there are

$$\left(\frac{n \cdot m - n}{m} \min_j\{\max\{c_j, p_j\}\}\right) \cdot \lambda$$

possible values for M. Hence, the total number of steps of the binary search is

$$O\left(\log\left(\frac{n \cdot m - n}{m} \min_j\{\max\{c_j, p_j\}\}\right) + \log(\lambda)\right).$$

Now we have to prove that this is polynomial in the size of our problem input.

Our platform parameters c_i and p_i are given under the form $p_i = \frac{\alpha_i}{\beta_i}$ and $c_i = \frac{\gamma_i}{\delta_i}$. So it takes $\log(\alpha_i) + \log(\beta_i)$ to store a p_i and $\log(\gamma_i) + \log(\delta_i)$ to store a c_i. So our entry E can be bounded as follows:

$$|E| \geq \sum_i \log(\alpha_i) + \sum_i \log(\beta_i) + \sum_i \log(\gamma_i) + \sum_i \log(\delta_i).$$

We can do the following estimation:

$$|E| \geq \sum_i \log(\beta_i) + \sum_i \log(\delta_i) = \log\left(\prod_i \beta_i \times \prod_i \delta_i\right) \geq \log(\lambda).$$

So we already know that our complexity is bounded by

$$O\left(|E| + \log\left(\frac{n \cdot m - n}{m} \min_j\{\max\{c_j, p_j\}\}\right)\right).$$

We can simplify this expression:

$$O\left(|E| + \log\left(n \cdot \min_j\{\max\{c_j, p_j\}\} - \frac{n}{m} \min_j\{\max\{c_j, p_j\}\}\right)\right)$$
$$\leq O\left(|E| + \log\left(n \cdot \min_j\{\max\{c_j, p_j\}\}\right)\right).$$

It remains to upper-bound:

$$\log(n \cdot \min_j\{\max\{c_j, p_j\}\}) = \log(n) + \log(\min_j\{\max\{c_j, p_j\}\}).$$

n is a part of the input and hence its size can be upper-bounded by the size of the input E. In the same manner we can upper-bound $\log(\min_j\{\max\{c_j, p_j\}\})$ by $\min_j\{\max\{\log(\alpha_j) + \log(\beta_j), \log(\gamma_j) + \log(\delta_j)\}\} \leq E$.

Assembling all these upper-bounds, we get

$$O\left(|E| + \log\left(n \cdot \min_j\{\max\{c_j, p_j\}\}\right)\right) \leq O(3|E|)$$

and hence our proposed algorithm needs $O(|E|)$ steps to perform the binary search. Furthermore, the set S contains at most $m \cdot n$ elements and the complexity of each call to the Adapted Moore's algorithm is thus $O(m \cdot n \cdot \log(m \cdot n))$, which is polynomial in the size of our problem input[2]. The total complexity finally is $O(|E| \cdot n \cdot m \cdot \log(n \cdot m))$ which is polynomial in the input size. ■

We can see here a new line between NP-hard problems and easier problems, because the scheduling problem becomes NP-hard when the platform is a heterogeneous tree instead of a star [62].

With release dates

We will prove that the problem $Q, MS \mid r_i \ ; \ p_j \ ; \ c_j \mid C_{max}$ is NP-hard in the strong sense. For that purpose, we will study the following decision problem:

Definition 3.1. ***MS-hetero:*** *Given a fully heterogeneous master-worker platform composed of m workers, n identical tasks with release dates, and a deadline D, is it possible to schedule those tasks onto this platform such that all tasks are completed before the deadline D ?*

Those two problems are equivalent. If *MS-hetero* can be solved in polynomial time, then our problem could also be solved in polynomial time using a binary search on D on the interval $[\min_j\{\frac{n}{m} \times \max\{c_j, p_j\} + \min\{c_j, p_j\}\}, r_n + \min_j\{n \times \max\{c_j, p_j\} + \min\{c_j, p_j\}\}]$. Reciprocally, if we can solve our problem and find the minimum makespan D_{opt}, then for all $D \geq D_{opt}$, *MS-hetero* has a schedule, elsewhere it does not.

We practice here a reduction from the work of Dutot [62].

Definition 3.2 (MS-Dutot)**.** *We suppose a one-port model. Let $T = (V, E)$ be a tree, but not a fork graph. Let P_{-1} in V be a special vertex called "Master node". For each other vertex, let p_i be the computation cost. For all edges e_i in E, let c_i be the communication cost. Finally let n be a number of tasks and D be a deadline.*

The decision problem is: "Is it possible to schedule the n tasks before the deadline D"?

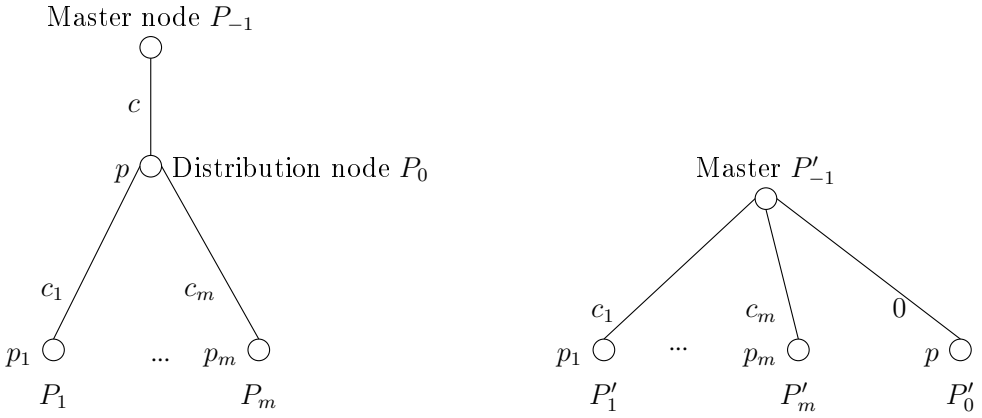
Theorem 3.6 ([62])**.** MS-Dutot *is NP-complete in the strong-sense.*

Theorem 3.7. MS-hetero *is NP-complete in the strong-sense.*

Proof. First, *MS-hetero* is in NP. If we have a schedule for n tasks and a given master-worker platform, we can check in polynomial time that this schedule respects the deadline D.

Let S be an instance of *MS-Dutot*. S is made of a tree T and of n tasks $J_1, ..., J_n$. We suppose that T is a two-level tree as illustrated in Figure 3.8, with one master node P_{-1}, one distribution node P_0, and m workers, P_1, ..., P_m. P_{-1} is only connected to P_0, and P_0 is connected to all other vertices (as the platform topology is a tree these are the only edges it contains). It takes a time p for P_0 to compute a task, and p_i for processor P_i, $1 \leq i \leq m$. It takes a time c for a task to be sent over the edge (P_{-1}, P_0) and, for $i \in [1, m]$, it takes a time c_i on the edge (P_0, P_i). We can assume without any loss of generality that any instance S has this shape as Dutot exactly used this kind of tree to build his proof of NP-completeness.

[2]We make the usual assumption that our scheduling problem takes the tasks 1, 2, ..., n as input, and not merely the number of identical tasks. This is a natural assumption which guarantees that basic certificates, such as sets of task starting times, are of size polynomial in the size of the input, and thus that scheduling problems are in NP.



Platform T used in *MS-Dutot*. Platform T' used in *MS-hetero*.

Figure 3.8: The platforms used in the reduction of *MS-Dutot* to *MS-hetero*.

From instance S of *MS-Dutot*, we build an instance S' of *MS-Hetero*, composed of a tree T' and n tasks, $J'_1, ..., J'_n$, with their respective release dates, $r'_1, ..., r'_n$ (Figure 3.8). T' is a fork tree, composed of one master P'_{-1} and $m+1$ leaves $P'_0, P'_1, .., P'_m$. The communication cost of the edge (P'_{-1}, P'_0) is 0 and, for each $i \in [1, m]$, the cost of the edge (P'_{-1}, P'_i) is c_i. The computation cost of P'_0 is p and, for each $i \in [1, m]$, the computation cost of P'_i is p_i. Finally, the release dates of the tasks are uniformly spaced: $\forall\ 1 \leq i \leq n, r_i = ic$. This reduction is obviously polynomial in the size of the original instance.

We now prove the equivalence between problems S and S':

- We first suppose that problem *MS-hetero* on S' has a solution, i.e., that there exists a schedule σ' which executes the n tasks on the $1+m$ processors, according to their release dates, and in an overall time less than or equal to D. From σ', we create a new schedule σ for S. Under σ, the master node P_{-1} sends all the tasks as soon as possible, the i-th task J_i being sent during the time interval $[(i-1)c, ic[$. Therefore, under σ a task J_i arrives on P_0 exactly at the release date of its corresponding task J'_i.

  For any $i \in [1, n]$, J_i is executed on P_j under σ if and only if J'_i is executed on P'_j under σ'. Furthermore, J_i is sent to P_j under σ at the date J'_i is sent to P'_j under σ', and the two corresponding tasks are executed at the same time. In particular, if J'_i was executed on P'_0, P_0 executes itself J_i without forwarding it to one of its workers.

  As σ' successfully schedules the n tasks with release dates J'_1, ..., J'_n on T' before the deadline D, σ schedules the n tasks J_1, ..., J_n on T before D.

- We now suppose that instance S of *MS-Dutot* has a solution, i.e., that there exists a schedule σ which executes the n tasks J_1, ..., J_n on T before the deadline D. As the time needed by the master P_{-1} to send tasks to the distribution node P_0 is c, task i will arrive on the distribution node at a time t_i greater than, or equal to, r_i.

  Then, symmetrically to what we have done in the previous case, we define σ' from σ as follows: for any $i \in [1, n]$, J'_i is executed on P'_j under σ' if and only if J_i is executed on P_j under σ, J_i is sent to P_j under σ at the date J'_i is sent to P'_j under σ', and the two corresponding tasks are executed at the same time.

We have noticed that, for any $i \in [1, n]$, $t_i \geq r_i$, therefore schedule σ' respects the release dates. As it has the same makespan than σ, we can conclude.

In conclusion, *MS-hetero* is NP-complete in the strong sense. ■

3.4 MPI experiments

To complement the previous theoretical results, we looked at some efficient online algorithms, and we compared them experimentally on different kinds of platforms. In particular, we include in the comparison our last two new heuristics, which were specifically designed to work well on communication-homogeneous and on computation-homogeneous platforms respectively.

3.4.1 The algorithms

We describe here the different algorithms used in the practical tests:

1. ***DD*** (*Demand Driven*) is a well known algorithm: each time a processor is free of work, it asks the master for a new task.
2. ***LS*** (*List Scheduling*) is an efficient algorithm on stable platforms. It uses its knowledge of the system (estimated computation time and communication time) to send a task as soon as possible to the worker that would finish it first, according to the current load estimation (the number of tasks already waiting for execution on the worker).
3. ***RR*** (*Round Robin*) is the simplest algorithm. It sends a task to each worker one by one, according to a cyclic fashion: it will send task i to processor i mod p and will send it as soon as possible
4. ***RRC*** is a variant of *RR* performing some resource selection: it sends the tasks starting from the worker with the smallest c_i up to the worker with the largest one until the first processor finishes its computation or until all processors receive one task, and then repeat this procedure again.
5. ***RRP*** has a similar behavior as *RRC*, but looks into computation instead of communication: it sends the tasks starting from the worker with the smallest p_i up to the worker with the largest one, until the first processor finishes its task or until all processor receive one task, and then repeat this procedure again.
6. ***SLJF*** is described in a previous section. It is optimal for makespan minimization on communication-homogeneous platform as soon as it knows the total number of tasks.
7. ***SLJFWC*** is our heuristic meant to be used on computation-homogeneous platform.

3.4.2 The experimental platform

During the experiments, we built a small real heterogeneous master-worker platform with five different laptops running the Linux operating system and connected to each other by a fast Ethernet switch (100 MB/s). The five machines are all different, both in terms of the amount of available memory and in terms of CPU speed (four have a CPU speed between 1.2 and 1.6 GHz, the last and oldest one has a processor Mobile Pentium MMX 233 MHz). The heterogeneity of

the communication links is mainly due to the differences between the network cards (the oldest and slowest machine has a 10 MB/s network card, the four others have 100 MB/s ones). From this platform, we selected one machine to be the master.

3.4.3 The tasks

Each task is composed of a matrix, and each worker has to calculate the determinant of the matrices it receives. Whenever needed, we play with matrix sizes so as to achieve more heterogeneity or on the contrary some homogeneity in the CPU speeds or communication bandwidths. We proceed as follows: in a first step, we send one single matrix to each worker one after the other, and we calculate the time needed to send this matrix and to calculate its determinant on each worker. Thus, we obtain an estimation of c_i and p_i, according to the matrix size. Then we determine the number of times this matrix should be sent (n_{c_i}) and the number of times its determinant should be calculated (n_{p_i}) on each worker in order to modify the platform characteristics so as to reach the desired level of heterogeneity. Then, a task (matrix) assigned on P_i will actually be sent n_{c_i} times to P_i (so that $c_i \leftarrow n_{c_i}.c_i$), and its determinant will actually be calculated n_{p_i} times by P_i (so that $p_i \leftarrow n_{p_i}.p_i$).

Here are the original values of c_i and p_i for a small matrix (40×40):

- $c_1 = 0.011423$ and $p_1 = 0.052190$
- $c_2 = 0.012052$ and $p_2 = 0.019685$
- $c_3 = 0.016808$ and $p_3 = 0.101777$
- $c_4 = 0.043482$ and $p_4 = 0.288397$

The experiments are as follows: for each diagram, we create ten random platforms, possibly with one prescribed property (such as homogeneous links or processors) and we execute the different algorithms on it. After modification of the parameters, our platforms were composed of machines P_i with c_i between 0.01 s and 1 s, and p_i between 0.1 s and 8 s. Once the platform is created, we send one thousand tasks to it, their release dates following a Poisson's distribution. Then we calculate the makespan, sum-flow, and max-flow obtained by each algorithm. After having executed all algorithms on the ten platforms, we calculate the average makespan, sum-flow, and max-flow. The following section shows the different results that have been obtained.

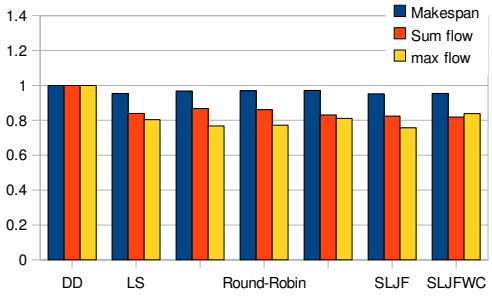
3.4.4 Results

For each algorithm we plot its makespan (in blue), sum-flow (in orange), and max-flow (in yellow), which are represented in this order, from left to right. We normalize everything to the performance of ***DD***, whose makespan, max-flow and sum-flow are therefore set equal to 1.

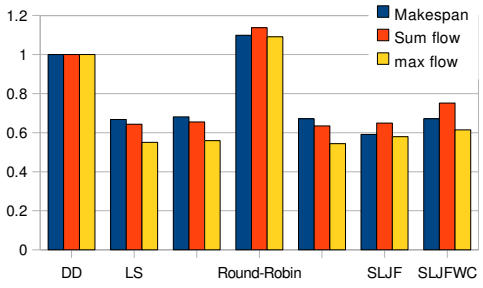
First of all, we consider fully homogeneous platforms. Figure 3.10(a) shows that all static algorithms perform equally well on such platforms, and exhibit better performance than the dynamic heuristic **DD**. On communication-homogeneous platforms (Figure 3.10(b)), we see that ***RRC***, which does not take processor heterogeneity into account, performs significantly worse than the others; we also observe that ***SLJF*** is the best approach for makespan minimization. On computation-homogeneous platforms (Figure 3.10(c)), we see that ***RRP*** and ***SLJF***, which do not take communication heterogeneity into account, perform significantly worse than the others; we also observe that ***SLJFWC*** is the best approach for makespan minimization. Finally, on fully heterogeneous platforms (Figure 3.10(d)), the best algorithms are ***LS*** and ***SLJFWC***. Moreover, we see that algorithms taking communication delays into account actually perform better. We underline here that taking into account the communication heterogeneity is more

important than the computation heterogeneity for a scheduler. This could be explained mainly because of the one-port model.

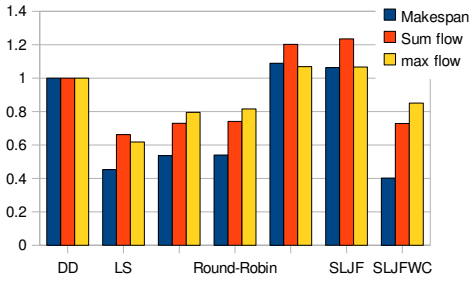
Figure 3.9: Comparison of the seven algorithms on different platforms.



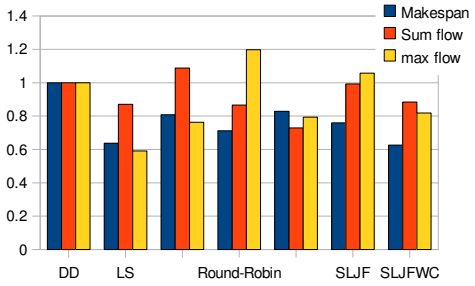
(a) Homogeneous platforms



(b) Platforms with homogeneous communication links



(c) Platforms with homogeneous processors



(d) Fully heterogeneous platforms

3.5 Related work

Task graph scheduling– Task graph scheduling is usually studied using the so-called *macro-dataflow* model (Section 2.4.2). In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units. More recent papers [138, 111, 121, 16, 22, 134] take communication resources into account.

Hollermann et al. [77] and Hsu et al. [81] introduce the following model for task graph scheduling: each processor can either send or receive a message at a given time-step (bidirectional communication is not possible); also, there is a fixed latency between the initiation of the communication by the sender and the beginning of the reception by the receiver. Still, the model is rather close to the one-port model discussed here.

Online scheduling– A good survey of online scheduling can be found in [127, 118]. Two papers focus on the problem of online scheduling for master-worker platforms. In [100],

Leung and Zhao proposed several competitive algorithms minimizing the total completion time on a master-worker platform, with or without pre- and post-processing. In [99], the same authors studied the complexity of minimizing the makespan or the total flow time, and proposed some heuristics. However, none of these works take into consideration communication costs. To the best of our knowledge, the first known results for online problems with communication costs are those reported in our former work [113], and in [B2] we dramatically improved several of the competitive ratios given in [113] and we have added new ones.

Master-worker on a computational grid – Master-worker scheduling on a grid can be based on a network-flow approach [129, 128] or on an adaptive strategy [75]. Note that the network-flow approach of [129, 128] is possible only when using a full multiple-port model, where the number of simultaneous communications for a given node is not bounded. This approach has also been studied in [78]. Enabling frameworks to facilitate the implementation of master-worker tasking are described in [72, 144].

3.6 Conclusion

In this chapter, we have dealt with the problem of online scheduling independent, same-size tasks on master-worker platforms. We enforce the one-port model, and we study the impact of heterogeneity on the performance of online and offline scheduling algorithms as well as the impact of the communications on the design and analysis of the proposed algorithms.

The major contribution of this work lies on the theoretical side, and is well summarized by Table 3.1 for online scheduling. We have provided a comprehensive set of lower bounds for the competitive ratio of any deterministic online scheduling algorithm, for each source of heterogeneity and for each target objective function. We also have derived several new results for offline scheduling with release dates, as an optimal makespan-minimization algorithm for communication-homogeneous platform, and an NP-hard proof of the scheduling problem on fully heterogeneous platform. Another important direction for future work would be to derive an optimal algorithm or to prove the NP-hardness for offline scheduling with release dates on computation-homogeneous platforms. On the practical side, we have to widen the scope of the MPI experiments. So we hope to have demonstrated in this chapter the superiority of those heuristics which fully take into account the relative capacity of the communication links.

Chapter 4

Matrix product

In this chapter, we expand our scheduling approach to the case of matrix' multiplication. Our goal is to minimize the total execution time, which remains a difficult problem considering our framework. In order to bypass this difficulty, we will only concentrate into minimizing the total communication volume, which we think is the bottleneck of this application.

4.1 Introduction

Matrix product is a key computational kernel in many scientific applications, and it has been extensively studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [41] and the ScaLAPACK outer product algorithm [33]. Typically, parallel implementations work well on 2D processor grids, because the input matrices are sliced horizontally and vertically into square blocks that are mapped one-to-one onto the physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor grids.

As we target heterogeneous clusters, which are composed of heterogeneous computing resources interconnected by a *sparse* network, we can no longer suppose there are direct links between any pair of workers. That means that messages from one processor to another are routed via several links, likely to have different capacities, and congestion will occur when two messages, involving two different sender/receiver pairs, collide because a same physical link happens to belong to the two routing paths. Therefore, an accurate estimation of the communication cost requires a precise knowledge of the underlying target platform.

There are two possible approaches to tackle the parallelization of matrix product on heterogeneous clusters when aiming at reusing the 2D processor grid strategy. The first (drastic) approach is to *ignore* communications. The objective is then to load-balance computations as evenly as possible on a heterogeneous 2D processor grid. This corresponds to arranging the n available resources as a (virtual) 2D grid of size $p \times q$ (where $p \cdot q \leq n$) so that each processor receives a share of the work, i.e., a rectangle, whose area is proportional to its relative computing speed. There are many processor arrangements to consider, and determining the optimal one is a highly combinatorial problem, which has been proven NP-complete in [12]. In fact, because of the geometric constraints imposed by the 2D processor grid, a perfect load-balancing can only be achieved in some very particular cases.

The second approach is to relax the geometric constraints imposed by a 2D processor grid.

The idea is then to search for a 2D partitioning of the input matrices into rectangles that will be mapped one-to-one onto the processors. Because the 2D partitioning now is irregular (it is no longer constrained to a 2D grid), some processors may well have more than four neighbors. The advantage of this approach is that a perfect load-balancing is always possible; for instance partitioning the matrices into horizontal slices whose vertical dimension is proportional to the computing speed of the processors always leads to a perfectly balanced distribution of the computations. The objective is then to minimize the total cost of the communications. However, it is very hard to accurately predict this cost. Indeed, the processor arrangement is virtual, not physical: as explained above, the underlying interconnection network is not expected to be a complete graph, and communications between neighbor processors in the arrangement are likely to be realized via several physical links constituting the communication path. The actual repartition of the physical links across all paths is hard to predict, but contention is almost certain to occur. This is why a natural, although pessimistic assumption, to estimate the communication cost, is to assume that all communications in the execution of the algorithm will be implemented sequentially. With this hypothesis, minimizing the total communication cost amounts to minimizing the total communication volume. Unfortunately, this problem has been shown NP-complete as well [13]. Note that even under the optimistic assumption that all communications at a given step of the algorithm can take place in parallel, the problem remains NP-complete [15].

In this chapter, we are not interested in adapting 2D processor grid strategies to heterogeneous clusters, as proposed in [87, 12, 13]. Instead, we adopt a realistic application scenario, where input files are read from a fixed repository (such as a disk on a data server). Computations will be delegated to available resources in the target architecture, and results will be returned to the repository, which calls for a master-worker paradigm. In this centralized approach, all matrix files originate from, and must be returned to, the master. The master distributes both data and computations to the workers (while in ScaLAPACK, input and output matrices are supposed to be equally distributed among participating resources beforehand). Typically, our approach is useful in the context of speeding up MATLAB or SCILAB clients running on a server (which acts as the master and initial repository of files).

In addition, it becomes necessary to include the cost of both the initial distribution of matrices to processors and of collecting back results. These input/output operations have always been neglected in the analysis of the conventional algorithms. This is because only $\Theta(n^2)$ coefficients need to be distributed in the beginning, and gathered at the end, as opposed to the $\Theta(n^3)$ computations[1] to be performed (where n is the problem size). The assumption that these communications can be ignored could have made sense on dedicated processor grids like, say, the Intel Paragon, but it is no longer reasonable on heterogeneous platforms. Furthermore, because we investigate the parallelization of large problems, we cannot assume that full matrix panels can be stored in worker memories and re-used for subsequent updates (as in ScaLAPACK). As processors cannot store all the matrices in their memory, the total required volume of communication can be larger than $\Theta(n^2)$ as a same matrix element may have to be sent several times to a same processor.

To summarize, the target platform is composed of several workers with different computing powers, different bandwidth links to/from the master, and different, limited, memory capacities. The first problem is *resource selection*. Which workers should be enrolled in the execution?

[1] Of course, there are $\Theta(n^3)$ computations if we only consider algorithms that use the standard way of multiplying matrices; this excludes Strassen's and Winograd's algorithms [56].

All of them, or maybe only the faster computing ones, or else only the faster-communicating ones? Once participating resources have been selected, there remain several scheduling decisions to be taken: how to minimize the number of communications? in which order workers should receive input data and return results? what amount of communications can be overlapped with (independent) computations? The goal of this work is to design efficient algorithms for resource selection and communication ordering. In addition, we report numerical experiments on various heterogeneous platforms at the École Normale Supérieure de Lyon and at the University of Tennessee.

The main contributions are twofold:

- *On the theoretical side*, we have refined the existing bounds on the volume of communications needed to perform a matrix-product on a platform with insufficient memory to simultaneously store the whole three matrices.
- *On the practical side*, we have designed an algorithm for heterogeneous platforms which is quicker than the existing ones, and which uses less computational resources, according to our MPI experiments.

The rest of this section is organized as follows. In Section 4.2, we state the scheduling problem precisely, and we introduce some notations. Next, we show that minimizing the execution time is still a difficult problem with more than one worker (Section 4.3). In Section 4.4, we proceed with the analysis of the total communication volume that is needed in the presence of memory constraints. We improve a well-known bound by Toledo et al. [140, 82] and we propose an algorithm almost achieving this bound on platforms with a single worker (Section 4.4.2). We deal with homogeneous platforms in Section 4.5, and we extend our algorithm to matrix multiplication on heterogeneous platforms in Section 4.6, and to LU factorization in Section 4.7. We report on some MPI experiments in Section 4.8. Finally, we give an overview of the related work in Section 4.9, and we conclude in Section 4.10.

4.2 Framework

In this section we formally state our hypotheses on the application (Section 4.2.1) and on the target platform (Section 4.2.2).

4.2.1 Application

We deal with the computational kernel $\mathcal{C} \leftarrow \mathcal{C} + \mathcal{A} \times \mathcal{B}$. We partition the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ as illustrated in Figure 4.1. Formally:

- We use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square *blocks* of size $q \times q$ (hence with q^2 coefficients). The size q of these square blocks is chosen so as to harness the power of Level 3 BLAS routines [32, 33]. Typically, $q = 80$ or 100 on most platforms when using ATLAS-generated routines [55, 145].
- The input matrix $\mathcal{A}$ is of size $n_{\mathcal{A}} \times n_{\mathcal{AB}}$:
  - we split $\mathcal{A}$ into r horizontal stripes $\mathcal{A}_i$, $1 \leq i \leq r$, where $r = n_{\mathcal{A}}/q$;
  - we split each stripe $\mathcal{A}_i$ into t square $q \times q$ blocks $\mathcal{A}_{i,k}$, $1 \leq k \leq t$, where $t = n_{\mathcal{AB}}/q$.

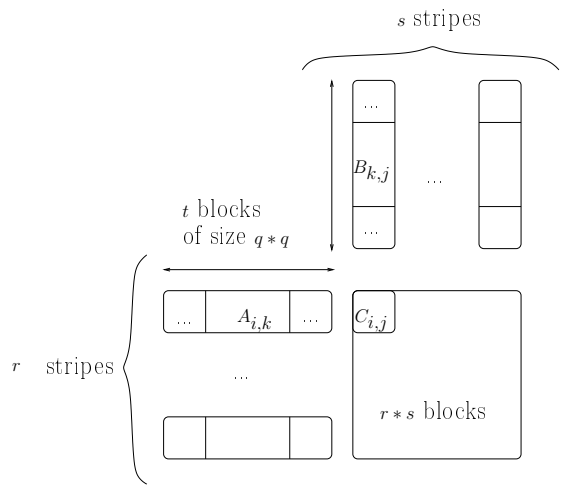


Figure 4.1: Partition of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

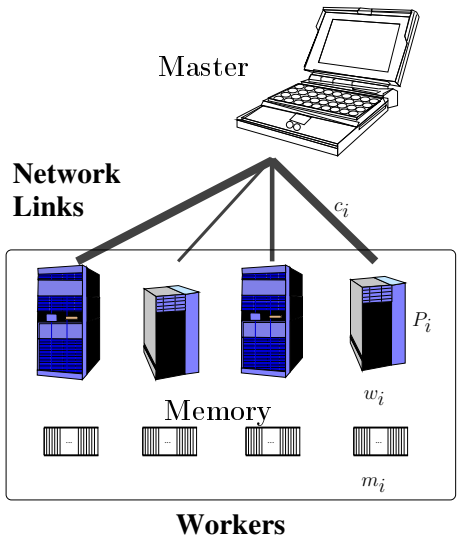


Figure 4.2: A fully heterogeneous master-worker platform with limited memory.

- The input matrix $\mathcal{B}$ is of size $n_{\mathcal{AB}} \times n_{\mathcal{B}}$:
  - we split $\mathcal{B}$ into s vertical stripes $\mathcal{B}_j$, $1 \leq j \leq s$, where $s = n_{\mathcal{B}}/q$;
  - we split stripe $\mathcal{B}_j$ into t square $q \times q$ blocks $\mathcal{B}_{k,j}$, $1 \leq k \leq t$.
- We compute $\mathcal{C} = \mathcal{C} + \mathcal{A} \times \mathcal{B}$. Matrix $\mathcal{C}$ is accessed (both for input and output) by square $q \times q$ blocks $\mathcal{C}_{i,j}$, $1 \leq i \leq r$, $1 \leq j \leq s$. There are $r \times s$ such blocks.

We point out that with such a decomposition, all stripes and blocks have the same size. This will greatly simplify the analysis of communication costs.

4.2.2 Platform

We still target a master-worker platform composed of p workers, connected by a star network ruled by the one-port model (see Figure 4.2). Because we manipulate large data blocks, we adopt a linear cost model, both for computations and communications (i.e., we neglect start-up overheads). We have the following notations:

Computation speed: It takes $X.w_i$ time-units to execute a task of size X on P_i;

Communication speed: It takes $X.c_i$ time-units for the master P_0 to send a message of size X to P_i or to receive a message of size X from P_i;

Memory capacity: the amount of memory available in each worker is expressed as a given number m_i of buffers, where a buffer can store one block of size $q \times q$ (either from $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$).

For large problems, this memory limitation will considerably impact the design of the algorithms, as data re-use will be greatly dependent on the amount of available buffers.

During most part of this chapter, we will suppose that the master has no processing capability. One can argue that if the master had processing capability, we could add a fictitious extra worker paying no communication cost to simulate computation at the master.

4.3 Combinatorial complexity of a simple version of the problem

This section is almost a digression; it is devoted to the study of the simplest variant of the problem. It is intended to show the intrinsic combinatorial difficulty of the problem. We make the following simplifications:

- We consider only rank-one block updates; in other words, and with previous notations, we focus on the case where $t = 1$ (so the matrices are of size $r \times 1, 1 \times s, r \times s$).
- Results need not be returned to the master.
- Workers have *no* memory limitation; they receive each stripe only once and can re-use them for other computations.

There are five parameters in the problem; three platform parameters (c, w, and the number of workers p) and two application parameters (r and s). The scheduling problem amounts to deciding which files should be sent to which workers and in which order. A given file may well be sent several times, to further distribute computations. For instance, a simple strategy is to partition $\mathcal{A}$ and to duplicate $\mathcal{B}$, i.e., send each block $\mathcal{A}_i$ only once and each block $\mathcal{B}_j$ p times; all workers would then be able to fully work in parallel.

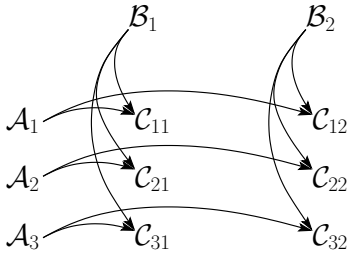


Figure 4.3: Dependence graph of the problem (with $r = 3$ and $s = 2$).

The dependence graph of the problem is depicted in Figure 4.3. It suggests a natural strategy for enabling workers to start computing as soon as possible. Indeed, the master should alternate sending of $\mathcal{A}$-blocks and $\mathcal{B}$-blocks. Of course it must be decided how many workers to enroll and in which order one needs to send the blocks to the enrolled workers. But with a single worker, we can show that the *alternating greedy* algorithm is optimal:

Proposition 4.1. *With a single worker, the* alternating greedy *algorithm is optimal.*

Proof. In this algorithm, the master sends blocks as soon as possible, alternating a block of type $\mathcal{A}$ and a block of type $\mathcal{B}$ (and proceeds with the remaining blocks when one type is exhausted). This strategy maximizes at each step the total number of tasks that can be processed by the worker. To see this, after x communication steps, with y files of type $\mathcal{A}$ sent, and z files of type $\mathcal{B}$ sent, where $y + z = x$, the worker can process at most $y \times z$ tasks. The greedy algorithm enforces $y = \lceil \frac{x}{2} \rceil$ and $z = \lfloor \frac{x}{2} \rfloor$ (as long as $\max(x, y) \leq \min(r, s)$, and then sends the remaining files), hence its optimality. ∎

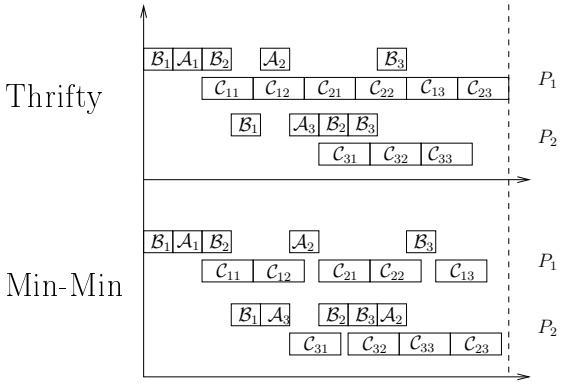


Figure 4.4: Example showing that Thrifty is not optimal: with $p = 2$, $c = 4$, $w = 7$, and $r = s = 3$, Min-min has a lower makespan. For each algorithm, the first line presents the communications for P_1, the second the computations of P_1, the third the communications for P_2, and the fourth the computations of P_2.

Unfortunately, for more than one worker, determining the next worker to enroll is a difficult problem. We believe that this general problem is NP-complete but we were unable to prove it. One can see the difficulty with the following example. There are (at least) two greedy algorithms that can be devised for p workers:

Thrifty: This algorithm "spares" resources as it aims at keeping each enrolled worker fully active. It works as follows:

- Send enough blocks to the first worker so that it is never idle,
- Send blocks to a second worker during spare communication slots, and
- Enroll a new worker (and send blocks to it) only if this does not delay previously enrolled workers.

Min-min: This algorithm is based on the well-known min-min heuristic [104]. At each step, all tasks are considered. For each of them, we compute their possible starting date on each worker, given the files that have already been sent to this worker and all decisions taken previously; we select the best worker, hence the first *min* in the heuristic. We take the minimum of starting dates over all tasks, hence the second *min*.

It turns out that neither greedy algorithm is optimal. See Figure 4.4 for an example where Min-min is better than Thrifty, and Figure 4.5 for an example of the opposite situation.

We now go back to our original model.

4.4 Minimization of the communication volume

Our underlying assumption is that communication costs dominate our problem. Therefore, we do not directly target makespan minimization, but communication volume minimization. Experiments, in Section 4.8, will show that our communication-volume minimization approach effectively leads to algorithms with shorter makespans.

In this section, we derive a lower bound on the total number of communications (sent from, or received by, the master) that are needed to execute any matrix multiplication algorithm

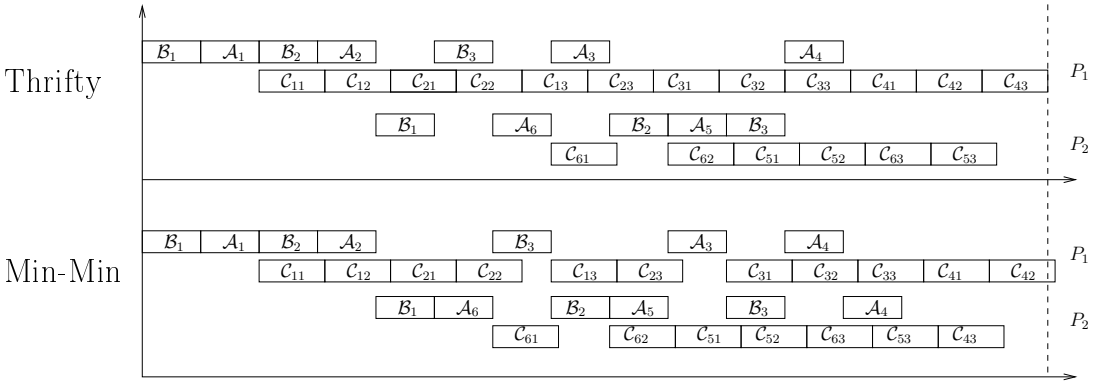


Figure 4.5: Example showing that Min-min is not optimal: with $p = 2$, $c = 8$, $w = 9$, $r = 6$, and $s = 3$, Thrifty has a lower makespan.

satisfying our hypotheses (centralized data and limited memory). Since we are under the one-port model, one can simulate the total communication volume of any parallel algorithm with a single worker, so we only need to consider the one-worker case.

Therefore, We deal with the following formulation of the problem:

- The master sends blocks $\mathcal{A}_{ik}$, $\mathcal{B}_{kj}$, and $\mathcal{C}_{ij}$,
- The master retrieves final values of blocks $\mathcal{C}_{ij}$,
- We enforce limited memory on the worker; only m buffers are available, which means that at most m blocks of $\mathcal{A}$, $\mathcal{B}$, and/or $\mathcal{C}$ can simultaneously be stored on the worker.

First, we improve the lower bound on the communication volume established by Toledo et al. [82]. Then, we describe an algorithm that aims at re-using $\mathcal{C}$ blocks as much as possible after they have been loaded, and we assess its performance.

4.4.1 Lower bound on the communication volume

To derive the lower bound, we refine an analysis due to Toledo [140]. The idea is to estimate the number of computations made thanks to m (the memory size) consecutive communication steps (once again, the unit here is a matrix block). We need some notations:

- We let α_{old}, β_{old}, and γ_{old} be the number of buffers used by blocks of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ right before the beginning of the m communication steps;
- We let α_{recv}, β_{recv}, and γ_{recv} be the number of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ blocks sent by the master during the m communication steps;
- Finally, we let γ_{send} be the number of $\mathcal{C}$ blocks returned to the master during these m steps.

Initially, the memory contains at most m blocks, and we consider a sequence of m communications. Therefore, the following equations must hold true:

$$\begin{cases} \alpha_{old} + \beta_{old} + \gamma_{old} \leq m \\ \alpha_{recv} + \beta_{recv} + \gamma_{recv} + \gamma_{send} = m \end{cases}$$

Toledo's inequality

The following lemma is given in [140]: consider any algorithm that uses the standard way of multiplying matrices (this excludes Strassen's and Winograd's algorithms [56], for instance). If N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$, and N_C elements of $\mathcal{C}$ are accessed, then no more than K computations can be done, where

$$K = \min\left\{(N_A + N_B)\sqrt{N_C}, (N_A + N_C)\sqrt{N_B}, (N_B + N_C)\sqrt{N_A}\right\}.$$

To use this result here, we remark that no more than $\alpha_{old}+\alpha_{recv}$ blocks of $\mathcal{A}$ are accessed, hence $N_A = (\alpha_{old} + \alpha_{recv})q^2$. Similarly, $N_B = (\beta_{old} + \beta_{recv})q^2$ and $N_C = (\gamma_{old} + \gamma_{recv})q^2$ (the $\mathcal{C}$ blocks returned are already counted). We simplify notations by writing:

$$\begin{cases} \alpha_{old} + \alpha_{recv} = \alpha m \\ \beta_{old} + \beta_{recv} = \beta m \\ \gamma_{old} + \gamma_{recv} = \gamma m \end{cases}$$

Then we obtain

$$K = \min\left\{(\alpha+\beta)\sqrt{\gamma}, (\beta+\gamma)\sqrt{\alpha}, (\gamma+\alpha)\sqrt{\beta}\right\} \times m\sqrt{m}q^3.$$

Writing $K = km\sqrt{m}q^3$, we obtain the following system of equations

$$\text{MAXIMIZE } k \text{ SUCH THAT}$$
$$\begin{cases} k \leq (\alpha + \beta)\sqrt{\gamma} \\ k \leq (\beta + \gamma)\sqrt{\alpha} \\ k \leq (\gamma + \alpha)\sqrt{\beta} \\ \alpha + \beta + \gamma \leq 2 \end{cases}$$

whose solution is easily found to be $\alpha = \beta = \gamma = \frac{2}{3}$, and $k = \sqrt{\frac{32}{27}}$. This gives a lower bound for the communication-to-computation ratio (in terms of blocks) of any algorithm:

$$\text{CCR}_{\text{opt}} = \frac{m}{km\sqrt{m}} = \sqrt{\frac{27}{32m}}.$$

Loomis-Whitney's inequality

In fact, it is possible to refine this bound. Instead of using the lemma given in [140], we use Loomis-Whitney inequality [82]: in any algorithm that uses the standard way of multiplying matrices, if N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$, and N_C elements of $\mathcal{C}$ are accessed, then no more than K computations can be done, where $K = \sqrt{N_A N_B N_C}$.
Here,

$$K = \sqrt{\alpha\beta\gamma} \times m\sqrt{m}q^3.$$

K is then maximized when $\alpha = \beta = \gamma = \frac{2}{3}$ and $k = \sqrt{\frac{8}{27}}$, so that the lower bound for the communication-to-computation ratio becomes: $\text{CCR}_{\text{opt}} = \sqrt{\frac{27}{8m}}$.

We point out that the bound CCR_{opt} improves upon the best-known value $\sqrt{\frac{1}{8m}}$ derived by Ironya, Toledo, and Tiskin [82].

4.4.2 The *maximum re-use* algorithm

In the above study, the lower-bound on the communication volume is obtained when the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are equally accessed during a sequence of communications. This may suggest to allocate one third of the memory to each of these matrices. In fact, Toledo [140] uses this memory layout. He even proves that, in the context of multiplication of square matrices of size r, his algorithm is "asymptotically optimal" as soon as the processor cannot store in its memory more than one sixth of one of the matrices: such an algorithm must have a communication-per-computation ratio which is $\Omega\left(\frac{r^3}{\sqrt{m}}\right)$ when his algorithm has a communication to computation ratio of $O\left(\frac{r^3}{\sqrt{m}}\right)$. We can, however, still significantly improve the performance of matrix multiplication in our context by reducing the constant hidden in the order of complexity of this ratio, as our experiments will show in Section 4.8.

A closer look at our problem shows that the multiplied matrices $\mathcal{A}$ and $\mathcal{B}$ have the same behavior, which differs from the behavior of the result matrix $\mathcal{C}$. Indeed, if an element of $\mathcal{C}$ is no longer used, it cannot be simply discarded from the memory as the elements of $\mathcal{A}$ and $\mathcal{B}$ are, but it must be sent back to the master. Intuitively, sending an element of $\mathcal{C}$ to a worker also costs the communication needed to retrieve it from the worker, and is thus twice as expensive as sending an element of $\mathcal{A}$ or $\mathcal{B}$.

Below we introduce and analyze the performance of the *maximum re-use* algorithm, that reuses as much as possible the elements of $\mathcal{C}$. Cannon's algorithm [41] and the ScaLAPACK outer product algorithm [33] both distribute square blocks of $\mathcal{C}$ to the processors. Intuitively, squares are better than elongated rectangles because their perimeter (which is proportional to the data needed by a worker to compute the area) is smaller for the same area. We use the same approach here.

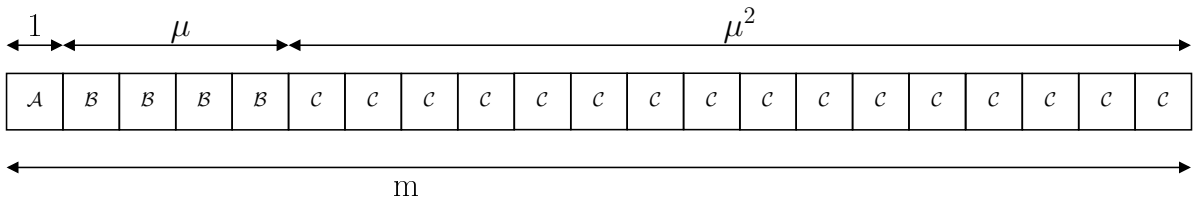


Figure 4.6: Memory layout for the *maximum re-use* algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.

The *maximum re-use* algorithm uses the memory layout illustrated in Figure 4.6. Four consecutive execution steps are shown in Figure 4.7. Assume that there are m available buffers. First we define μ as the largest integer such that $1 + \mu + \mu^2 \leq m$. The idea is to use one buffer to store $\mathcal{A}$ blocks, μ buffers to store $\mathcal{B}$ blocks, and μ^2 buffers to store $\mathcal{C}$ blocks. In the outer loop of the algorithm, a $\mu \times \mu$ square of $\mathcal{C}$ blocks is loaded. Once these μ^2 blocks have been loaded, they are repeatedly updated in the inner loop of the algorithm until their final value is computed. Then the blocks are returned to the master, and μ^2 new $\mathcal{C}$ blocks are sent by the master and stored by the worker. As illustrated in Figure 4.6, we need μ buffers to store a row of $\mathcal{B}$ blocks, but only one buffer for $\mathcal{A}$ blocks: $\mathcal{A}$ blocks are sent in sequence, each of them is used in combination with a row of μ $\mathcal{B}$ blocks to update the corresponding row of $\mathcal{C}$ blocks. This leads to the following sketch of the algorithm:

The performance of one iteration of the outer loop of the *maximum re-use* algorithm can

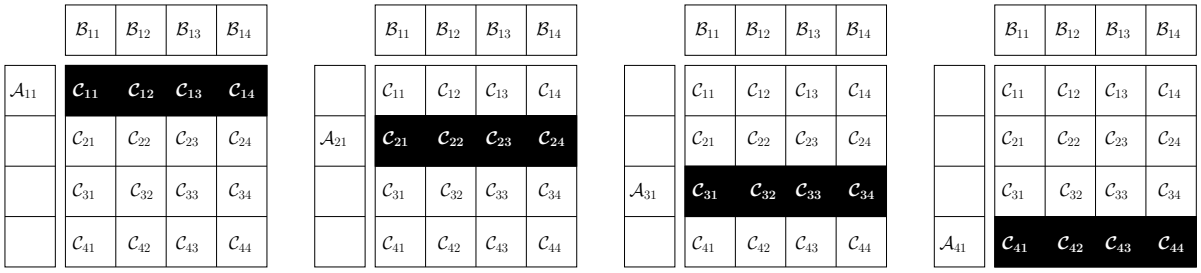


Figure 4.7: Four steps of the *maximum re-use* algorithm, with $m = 21$ and $\mu = 4$. Updated elements of $\mathcal{C}$ are written white on black.

Algorithm 5: The *maximum re-use* algorithm

while *there remain $\mathcal{C}$ blocks to be computed* **do**
  Receive μ^2 blocks of $\mathcal{C}$: $\{\mathcal{C}_{i,j} \mid i_0 \leq i < i_0 + \mu,\ j_0 \leq j < j_0 + \mu\}$;
  for $k \leftarrow 1$ **to** t **do**
    Receive a row of μ elements of $\mathcal{B}$ $\{\mathcal{B}_{k,j} \mid j_0 \leq j < j_0 + \mu\}$;
    Sequentially receive μ elements of one column of $\mathcal{A}$ $\{\mathcal{A}_{i,k} \mid i_0 \leq i < i_0 + \mu\}$;
    for *each* $\mathcal{A}_{i,k}$ **do**
      update μ elements of $\mathcal{C}$;
  Return results to master;

readily be determined:

- We need $2\mu^2$ communications to send and retrieve $\mathcal{C}$ blocks.
- For each value of t:
  - we need μ elements of $\mathcal{A}$ and μ elements of $\mathcal{B}$;
  - we update μ^2 blocks.

In terms of block operations, the communication-to-computation ratio achieved by the algorithm is thus

$$\text{CCR} = \frac{2\mu^2 + 2\mu t}{\mu^2 t} = \frac{2}{t} + \frac{2}{\mu}.$$

For large problems, i.e., large values of t, we see that the CCR is asymptotically close to the value $\text{CCR}_\infty = \frac{2}{\sqrt{m}}$. We point out that, in terms of data elements, the communication-to-computation ratio is divided by a factor q. Indeed, a block consists of q^2 coefficients but an update requires q^3 floating-point operations. Also, the ratio CCR_∞ achieved by the *maximum re-use* algorithm is lower by a factor $\sqrt{3}$ than the ratio achieved by the *blocked matrix-multiply* algorithm of [140].

Finally, we remark that the performance of the *maximum re-use* algorithm is quite close to the lower bound derived earlier: $\text{CCR}_\infty = \frac{2}{\sqrt{m}} = \sqrt{\frac{32}{8m}}$.

4.5 Algorithms for homogeneous platforms

In this section, we adapt the *maximum re-use* algorithm to fully homogeneous platforms. In this framework, the memory capacities of processors are limited. So we must first decide which part

of the memory will be used to stock which part of the original matrices, in order to maximize the total number of computations completed per time unit.

4.5.1 Principle of the algorithm

We load into the memory of each worker μ $q \times q$ blocks of $\mathcal{A}$ and μ $q \times q$ blocks of $\mathcal{B}$ to compute μ^2 $q \times q$ blocks of $\mathcal{C}$. In addition, we need 2μ extra buffers, split into μ buffers for $\mathcal{A}$ and μ for $\mathcal{B}$, in order to overlap computation and communication steps. In fact, μ buffers for $\mathcal{A}$ and μ for $\mathcal{B}$ would suffice for each update, but we need to prepare for the next update while computing. Overall, the number of $\mathcal{C}$ blocks that we can simultaneously load into memory is defined by the largest integer μ such that

$$\mu^2 + 4\mu \leq m.$$

We have to determine the number of participating workers, $\mathfrak{P}$. For that purpose, we proceed as follows. On the communication side, we know that in a round (computing a $\mathcal{C}$ block entirely), the master exchanges with each worker $2\mu^2$ blocks of $\mathcal{C}$ (μ^2 sent and μ^2 received), and sends μt blocks of $\mathcal{A}$ and μt blocks of $\mathcal{B}$. Also during this round, on the computation side, each worker computes $\mu^2 t$ block updates.

If we enroll too many processors, the communication capacity of the master will be exceeded: there is a limit on the number of blocks that it can send per time unit. On the contrary, if we enroll too few processors, they may be overloaded. We can compute the number of processors $\mathfrak{P}$ so that the time needed to send blocks to $\mathfrak{P}$ processors will be roughly equal to (or slightly greater than) the time spent by one processor for its computations. $\mathfrak{P}$ is the smallest integer such that

$$2\mu tc \times \mathfrak{P} \geq \mu^2 tw.$$

Indeed, this is the smallest value to saturate the communication capacity of the master required to sustain the corresponding computations. We derive that

$$\mathfrak{P} = \left\lceil \frac{\mu^2 tw}{2\mu tc} \right\rceil = \left\lceil \frac{\mu w}{2c} \right\rceil.$$

In the context of matrix multiplication, we have $c = q^2\tau_c$ and $w = q^3\tau_a$, where τ_c and τ_a respectively represent the speed of the communication link and the speed of the processor. Hence we have $\mathfrak{P} = \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil$. Moreover, we need to enforce that $\mathfrak{P} \leq p$, hence we finally obtain the formula

$$\mathfrak{P} = \min \left\{ p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil \right\}.$$

For the sake of simplicity, we suppose that r is divisible by μ, and s by $\mathfrak{P}\mu$. We allocate μ block columns (i.e., $q\mu$ consecutive columns of the original matrix) of $\mathcal{C}$ to each processor. The algorithm is made of two parts. Algorithm 6 outlines the program of the master, while Algorithm 7 is the program of each worker.

4.5.2 Impact of the start-up overhead

If we follow the execution of the homogeneous algorithm, we may wonder whether we can really neglect the input/output of $\mathcal{C}$ blocks. We sequentialize the sending, computing, and receiving of the $\mathcal{C}$ blocks, so that each worker loses $2c$ time-units per block, i.e., per tw time-units. As there are $\mathfrak{P} \leq \frac{\mu w}{2c} + 1$ workers, the total loss would be of $2c\mathfrak{P}$ time-units every tw time-units,

Algorithm 6: Homogeneous version, master program.

$\mu \leftarrow \lfloor \sqrt{4+m} - 2 \rfloor$;
$\mathfrak{P} \leftarrow \min \left\{ p, \lceil \frac{\mu w}{2c} \rceil \right\}$;
Split matrix $\mathcal{C}$ into squares $\mathbf{C}_{i',j'}$ of μ^2 $q \times q$ blocks :
$\mathbf{C}_{i',j'} = \{\mathcal{C}_{i,j} \mid (i'-1)\mu + 1 \leq i \leq i'\mu, (j'-1)\mu + 1 \leq j \leq j'\mu\}$;
for $j'' \leftarrow 0$ **to** $\frac{s}{\mathfrak{P}\mu}$ *by Step* $\mathfrak{P}$ **do**
  for $i' \leftarrow 1$ **to** $\frac{r}{\mu}$ **do**
    for $id_{worker} \leftarrow 1$ **to** $\mathfrak{P}$ **do**
      $j' \leftarrow j'' + id_{worker}$;
      Send block $\mathbf{C}_{i',j'}$ to worker id_{worker};
    for $k \leftarrow 1$ **to** t **do**
      for $id_{worker} \leftarrow 1$ **to** $\mathfrak{P}$ **do**
        $j' \leftarrow j'' + id_{worker}$;
        for $j \leftarrow (j'-1)\mu + 1$ **to** $j'\mu$ **do** Send $\mathcal{B}_{k,j}$;
        for $i \leftarrow (i'-1)\mu + 1$ **to** $i'\mu$ **do** Send $\mathcal{A}_{i,k}$;
    for $id_{worker} \leftarrow 1$ **to** $\mathfrak{P}$ **do**
      $j' \leftarrow j'' + id_{worker}$;
      Receive $\mathbf{C}_{i',j'}$ from worker id_{worker};

Algorithm 7: Homogeneous version, worker program.

for *all blocks* **do**
  Receive $\mathbf{C}_{i',j'}$ from master;
  for $k \leftarrow 1$ **to** t **do**
    for $j \leftarrow (j'-1)\mu + 1$ **to** $j'\mu$ **do** Receive $\mathcal{B}_{k,j}$;
    for $i \leftarrow (i'-1)\mu + 1$ **to** $i'\mu$ **do**
      Receive $\mathcal{A}_{i,k}$;
      for $j \leftarrow (j'-1)\mu + 1$ **to** $j'\mu$ **do** $\mathcal{C}_{i,j} \leftarrow \mathcal{C}_{i,j} + \mathcal{A}_{i,k}.\mathcal{B}_{k,j}$;
  Return $\mathbf{C}_{i',j'}$ to master;

which is less than $\frac{\mu}{t} + \frac{2c}{tw}$. For example, with $c = 2$, $w = 4.5$, $\mu = 4$ and $t = 100$, we enroll $\mathfrak{P} = 5$ workers, and the total loss is at most 4%, which is small enough to be neglected. Note that it would technically be possible to design an algorithm where the sending of the next block is overlapped with the last computations of the current block, but the whole procedure would get much more complicated.

4.5.3 Dealing with "small" matrices or platforms

We have shown that our algorithm should use $\mathfrak{P} = \min \left\{ p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil \right\}$ processors, each of them holding μ^2 blocks of matrix $\mathcal{C}$. For this solution to be feasible, $\mathcal{C}$ must be large enough. In other words, this solution can be implemented if and only if $r \times s \geq \min \left\{ p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil \right\} \mu^2$. If $\mathcal{C}$ is not

large enough, we will only use $\mathfrak{Q} < \mathfrak{P}$ processors, each of them holding ν^2 blocks of $\mathcal{C}$, such that:

$$\left\{ \begin{array}{l} \mathfrak{Q}\nu^2 \leq r \times s \\ \nu^2 w \leq 2\nu\mathfrak{Q}c \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} \mathfrak{Q}\nu^2 \leq r \times s \\ \frac{\nu w}{2c} \leq \mathfrak{Q} \end{array} \right. ,$$

following the same line of reasoning as previously. We obviously want ν to be the largest possible in order for the communications to be most beneficial. For a given value of ν we want $\mathfrak{Q}$ to be the smallest to spare resources. Therefore, the best solution is given by the largest value of ν such that:

$$\left\lceil \frac{\nu w}{2c} \right\rceil \nu^2 \leq r \times s,$$

and then $\mathfrak{Q} = \left\lceil \frac{\nu w}{2c} \right\rceil$.

If the platform does not contain the desired number of processors, i.e., if $\mathfrak{P} > p$ in the case of a "large" matrix $\mathcal{C}$ or if $\mathfrak{Q} > p$ otherwise, then we enroll all the p processors and we give them ν^2 blocks of $\mathcal{C}$ with $\nu = \min\left\{ \frac{r \times s}{p}, \frac{2c}{w} p \right\}$, following the same line of reasoning as previously.

4.6 Algorithms for heterogeneous platforms

We now consider the general problem, i.e., when processors are heterogeneous in terms of memory size as well as computation and/or communication times. As in the previous section, m_i is the number of $q \times q$ blocks that fit in the memory of worker P_i, and we need to load into the memory of P_i $2\mu_i$ blocks of $\mathcal{A}$, $2\mu_i$ blocks of $\mathcal{B}$, and μ_i^2 blocks of $\mathcal{C}$. This number of blocks loaded into memory changes from worker to worker, as it depends on their memory capacities: μ_i is the largest integer such that $\mu_i^2 + 4\mu_i \leq m_i$.

We first try to adapt our *maximum re-use* algorithm to heterogeneous platforms using a steady-state-like approach. But because of our model, we can easily discuss its limitations. We then introduce our final algorithm for heterogeneous platforms in Section 4.6.2.

4.6.1 Bandwidth-centric resource selection

Each worker P_i has parameters c_i, w_i, and μ_i, and each participating P_i needs a time $2\mu_i t c_i$ to receive $\delta_i = 2\mu_i t c_i$ blocks to perform $\phi_i = t\mu_i^2 w_i$ computations. Once again, we neglect I/O for $\mathcal{C}$ blocks. Let us consider the steady-state of a schedule. During one time-unit, P_i receives a certain amount y_i of blocks, both of $\mathcal{A}$ and $\mathcal{B}$, and computes x_i $\mathcal{C}$ blocks. We express the constraints, in terms of communications –the master has limited bandwidth– and of computations –a worker cannot perform more work than it receives. The objective is to maximize the amount of work performed per time-unit. Altogether, we gather the following linear program:

$$\left\{ \begin{array}{l} \text{MAXIMIZE } \sum_i x_i \\ \text{SUBJECT TO} \\ \sum_i y_i c_i \leq 1 \\ \forall i, \quad x_i w_i \leq 1 \\ \forall i, \quad \frac{x_i}{\mu_i^2} \leq \frac{y_i}{2\mu_i} \end{array} \right.$$

	P_1	P_2
c_i	1	x
w_i	2	$2x$
μ_i	2	2
$\frac{2c_i}{\mu_i w_i}$	$\frac{1}{2}$	$\frac{1}{2}$

Table 4.1: Platform for which the bandwidth-centric solution is not feasible.

Obviously, the best solution for y_i is $y_i = \frac{2x_i}{\mu_i}$, so the problem can be reduced to :

$$\begin{cases} \text{MAXIMIZE } \sum_i x_i \\ \text{SUBJECT TO} \\ \forall i, \quad x_i \leq \frac{1}{w_i} \\ \quad \sum_i \frac{2c_i}{\mu_i} x_i \leq 1 \end{cases}$$

The optimal solution for this system is a bandwidth-centric strategy [17, 9]: we sort workers by non-decreasing values of $\frac{2c_i}{\mu_i}$ and we enroll them as long as $\sum \frac{2c_i}{\mu_i w_i} \leq 1$. In this way, we can achieve the throughput $\rho \approx \sum_{i \text{ enrolled}} \frac{1}{w_i}$.

This solution seems to be close to the optimal. However, the problem is that workers may not have enough memory to execute it! Consider the example described by Table 4.1.

Using the bandwidth-centric strategy, every $8x$ seconds:

- P_1 receives $4x$ blocks (x $\mu_1 \times \mu_1$ chunks) in $4x$ seconds, and computes $4x$ blocks in $8x$ seconds;
- P_2 receives 4 blocks (1 $\mu_2 \times \mu_2$ chunk) in $4x$ seconds, and computes 4 blocks in $8x$ seconds.

But P_1 computes too quickly: during the time x needed to send a block to P_2, P_1 updates $\frac{x}{2}$ blocks, which requires at least x blocks and as many buffers. And x can be arbitrary large ! For example, if $x = 20$, P_1 would need buffers to store as many as 20 blocks to stay busy while one block is sent to P_2:

Communications	11111111111111111111	20	11111111111111111111	20	11111111...
Processor	P_1	P_2	P_1	P_2	P_1...

Therefore, the bandwidth-centric solution cannot always be realized in practice, and we turn to another algorithm, described below. To avoid the previous buffer problems, resource selection will be performed through a step-by-step simulation. However, we point out that the steady-state solution is an upper bound on the performance that can be achieved.

4.6.2 Incremental resource selection

The different memory capacities of the workers imply that we assign them chunks of different sizes. This requirement complicates the global partitioning of the $\mathcal{C}$ matrix among the workers. To take this into account and simplify the implementation, we decide to assign only full matrix column blocks in the algorithm. This is done in a two-phase approach.

In the first phase we pre-compute the allocation of blocks to processors, using a processor selection algorithm we will describe later. We start as if we had a huge matrix of size $\infty \times \sum_{i=1}^{n} \mu_i$.

Each time a processor P_i is chosen by the processor selection algorithm it is assigned a square chunk of μ_i^2 $\mathcal{C}$ blocks. As soon as some processor P_i has enough blocks to fill up μ_i block columns of the initial matrix, we decide that P_i will indeed execute these columns during the parallel execution. Therefore we maintain a panel of $\sum_{i=1}^{p} \mu_i$ block columns and fill them out by assigning blocks to processors. We stop this phase as soon as all the $r \times s$ blocks of the initial matrix have been allocated columnwise by this process. Note that worker P_i will be assigned a block column after it has been selected $\lceil \frac{r}{\mu_i} \rceil$ times by the algorithm.

In the second phase we perform the actual execution. Messages will be sent to workers according to the previous selection process. The first time a processor P_i is selected, it receives a square chunk of μ_i^2 $\mathcal{C}$ blocks, which initializes its repeated pattern of operation: the following t times, P_i receives μ_i $\mathcal{A}$ and μ_i $\mathcal{B}$ blocks, which requires $2\mu_i c_i$ time-units.

It remains to decide which processor to select at each step. We have no closed-form formula for the allocation of blocks to processors. Instead, we use an incremental algorithm to compute which worker the next blocks will be assigned to. We have two variants of the incremental algorithm, a *global* one that aims at optimizing the overall communication-to-computation ratio, and a *local* one that selects the best processor for the next stage. Both variants are described below.

Global selection algorithm

The intuitive idea, here, is to select the processor that maximizes the ratio of the total work achieved so far (in terms of block updates) over the completion time of the last communication. The latter represents the time spent by the master so far, either sending data to workers or staying idle, waiting for the workers to finish their current computations. We have:

$$\mathsf{ratio} \leftarrow \frac{\text{total work achieved}}{\text{completion time of last communication}}.$$

Estimating computations is easy: P_i executes μ_i^2 block updates per assignment. Communications are slightly more complicated to deal with; we cannot just use the communication time $2\mu_i c_i$ of P_i for the $\mathcal{A}$ and $\mathcal{B}$ blocks because we need to take its ready time into account. Indeed, if P_i is currently busy executing work, it cannot receive additional data too much in advance because its memory is limited. Algorithm 8 presented in this section presents this selection process, which we iterate until all blocks of the initial matrix are assigned and computed. We illustrate it on an example.

Running the global selection algorithm on an example. Consider the example described in Table 4.2 with three workers P_1, P_2 and P_3. For the first step, we have $\mathsf{ratio}_i \leftarrow \frac{\mu_i^2}{2\mu_i c_i}$ for all i. We compute $\mathsf{ratio}_1 = 1.5$, $\mathsf{ratio}_2 = 3$, and $\mathsf{ratio}_3 = 1$ and select P_2: $\mathsf{next} \leftarrow 2$. We update variables as $\mathsf{total\text{-}work} \leftarrow 0{+}324 = 324$, $\mathsf{completion\text{-}time} \leftarrow \max(0{+}108, 0) = 108$, $\mathsf{ready}_2 \leftarrow 108{+}972 = 1080$ and $\mathsf{nb\text{-}block}_2 \leftarrow 36$.

At the second step we compute $\mathsf{ratio}_1 \leftarrow \frac{324+36}{108+24} = 2.71$, $\mathsf{ratio}_2 \leftarrow \frac{324+324}{1080} = 0.6$ and $\mathsf{ratio}_3 \leftarrow \frac{324+100}{108+100} = 2.04$ and we select P_1. We point out that P_2 is busy until time $t = 1080$ because of the first assignment, which we correctly took into account when computing ready_2. For P_1 and P_3 the communication could take place immediately after the first one. It remains to update the variables: $\mathsf{total\text{-}work} \leftarrow 324 + 36 = 360$, $\mathsf{completion\text{-}time} \leftarrow \max(108 + 24, 0) = 132$, $\mathsf{ready}_1 \leftarrow 132 + 72 = 204$ and $\mathsf{nb\text{-}block}_1 \leftarrow 12$.

Algorithm 8: Global selection algorithm.

Data:
completion-time: completion time of the last communication
ready_i: ready time of processor P_i
$\mathsf{nb\text{-}block}_i$: number of A and B blocks sent to processor P_i
total-work: total work assigned so far (in terms of block updates)
nb-column: number of fully processed $\mathcal{C}$ block columns

INITIALIZATION

$\mathsf{completion\text{-}time} \leftarrow 0$;
$\mathsf{total\text{-}work} \leftarrow 0$;
for $i \leftarrow 1$ **to** p **do**
  $\mathsf{ready}_i \leftarrow 0$;
  $\mathsf{nb\text{-}block}_i \leftarrow 0$;

SIMULATION

repeat
  $\mathsf{next} \leftarrow$ worker that realizes $\max_{i=1}^{p} \frac{\mathsf{total\text{-}work}+\mu_i^2}{\max(\mathsf{completion\text{-}time}+2\mu_i c_i, \mathsf{ready}_i)}$;
  $\mathsf{total\text{-}work} \leftarrow \mathsf{total\text{-}work} + \mu_{\mathsf{next}}^2$;
  $\mathsf{completion\text{-}time} \leftarrow \max\{\mathsf{completion\text{-}time} + 2\mu_{\mathsf{next}} c_{\mathsf{next}}, \mathsf{ready}_{\mathsf{next}}\}$;
  $\mathsf{ready}_{\mathsf{next}} \leftarrow \mathsf{completion\text{-}time} + \mu_{\mathsf{next}}^2 w_{\mathsf{next}}$;
  $\mathsf{nb\text{-}block}_{\mathsf{next}} \leftarrow \mathsf{nb\text{-}block}_{\mathsf{next}} + 2\mu_{\mathsf{next}}$;
  $\mathsf{nb\text{-}column} \leftarrow \sum_{i=1}^{p} \left\lfloor \frac{\mathsf{nb\text{-}block}_i}{2\mu_i t \lceil \frac{r}{\mu_i} \rceil} \right\rfloor \mu_i$;
until *nb-column* $\geq s$;

At the third step the algorithm selects P_3. Going forward, we have a cyclic pattern repeating, with 13 consecutive communications, one to P_2 followed by 12 ones alternating between P_1 and P_3, and then some idle time before the next pattern (see Figure 4.8). The asymptotic value of ratio is 1.17 while the steady-state approach of Section 4.6.1 would achieve a ratio of 1.39 without taking memory limitations into account.

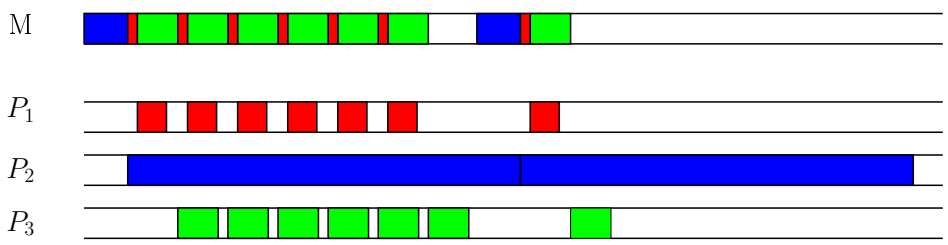


Figure 4.8: Global selection algorithm on the example of Table 4.2.

Finally, we point out that it is easy to further refine the algorithm to get closer to the performance of the steady-state. For instance, instead of selecting the best processor greedily, we could look two-steps ahead and search for the best pair of workers to select for the next two communications (the only price to pay is an increase in the cost of the selection algorithm). For the example, the two-block look-ahead strategy achieves an asymptotic ratio up to 1.30.

	P_1	P_2	P_3
c_i	2	3	5
w_i	2	3	1
μ_i	6	18	10
μ_i^2	36	324	100
$2\mu_i c_i$	24	108	100

Table 4.2: Platform used to demonstrate the processor selection algorithms.

Local selection algorithm

The global selection algorithm picks, as the next processor, the one that maximizes the ratio of the total amount of work assigned over the time needed to send all the required data. On the other hand, the local selection algorithm chooses, as destination of the i-th communication, the processor that maximizes the ratio of the amount of work assigned by this communication over the time during which the communication link is used to performed this communication (i.e., the elapsed time between the end of $(i-1)$-th communication and the end of the i-th communication). As previously, if processor P_j is the target of the i-th communication, the i-th communication is the sending of μ_j blocks of $\mathcal{A}$ and μ_j blocks of $\mathcal{B}$ to processor P_j, which enables it to perform μ_j^2 updates.

More formally, the local selection algorithm picks the worker P_i that maximizes:

$$\frac{\mu_i^2}{\max\{2\mu_i c_i,\ \mathsf{ready}_i - \mathsf{completion\text{-}time}\}}$$

Once again we consider the example described in Table 4.2. For the first three steps, the global and local selection algorithms make the same decisions. In fact, they take the same first 13 decisions. However, for the 14-th selection, the global algorithm picks processor P_2 when the local selection selects processor P_1 and then processor P_2 for the 15-th decision, as illustrated in Figure 4.9. Under both selection processes, its second chunk of work is sent to processor P_2 at the same time but the local algorithm inserts an extra communication. For this example, the local selection algorithm achieves an asymptotic ratio of computation per communication of 1.21. This is better than what is achieved by the global selection algorithm but, obviously, there are examples where the global selection will beat the local one.

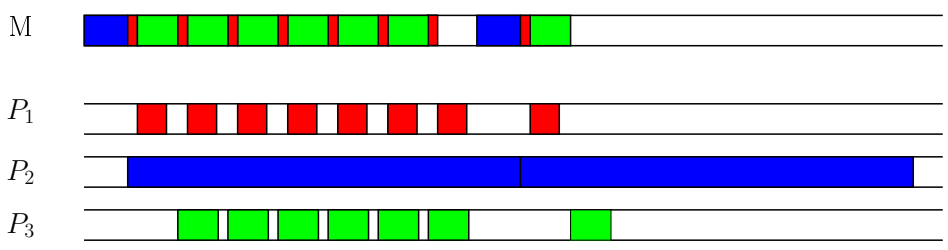


Figure 4.9: Local selection algorithm on the example of Table 4.2.

Variants

We have just presented a global and a local selection process. Each process can either take a decision only looking at the next communication, or can consider the next two communications, in a look-ahead approach. Furthermore, in the above selection processes we have not taken into account the cost of initially sending μ_i^2 blocks of matrix $\mathcal{C}$ to processor P_i the first time it receives blocks (or the cost of sending back μ_i^2 blocks of $\mathcal{C}$ to the master after t iterations, and then receiving μ_i^2 brand new blocks). We can take these costs into account in a variant: this would seem more realistic, but this may wrongly forbid to enroll an additional processor due to a huge initialization cost. Hence, we end up with 8 different selection algorithms (global or local, look-ahead or not, μ_i^2 $\mathcal{C}$ costs or not). There is no reason that one of these heuristics always dominate the others. We will thus consider the eight of them in our experiments.

4.7 Extension to LU factorization

In this section, we show how our techniques can be extended to LU factorization. We first consider the case of a single worker (Section 4.7.1), in order to study how we can minimize the communication volume. Then we present algorithms for homogeneous clusters (Section 4.7.2) and for heterogeneous platforms (Section 4.7.3).

We consider the right-looking version of the LU factorization as it is more amenable to parallelism. As previously, we use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square *blocks* of size $q \times q$ (hence with q^2 coefficients). The size of the matrix is then $r \times r$ blocks. Furthermore, we consider a second level of blocking of size μ. As previously, μ is the largest integer such that $\mu^2 + 4\mu \leq m$. The main kernel is then a rank-μ update $C \leftarrow C + A.B$ of blocks. Hence the similarity between matrix multiplication and LU decomposition.

4.7.1 Single processor case

The different steps of LU factorization are presented in Figure 4.10. Step k of the factorization consists of the following:

1. Factor pivot matrix (Figure 4.11(a)). We compute at each step a pivot matrix of size μ^2 (which thus contains $\mu^2 \times q^2$ coefficients). This factorization has a communication cost of $2\mu^2 c$ (to bring the matrix and send it back after the update) and a computation cost of $\mu^3 w$.

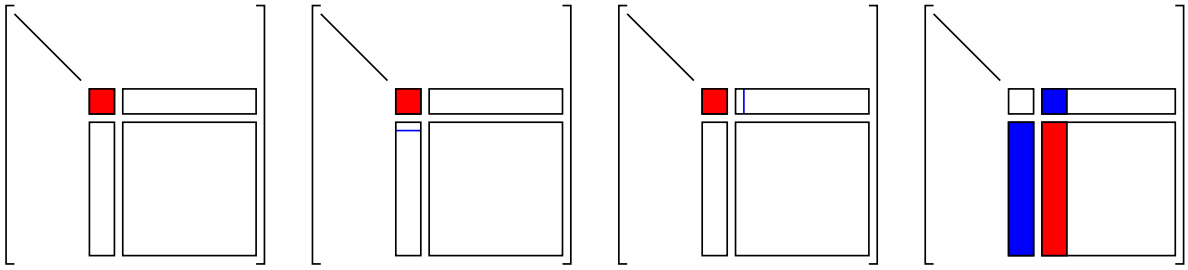
2. Update the μ columns below the pivot matrix (vertical panel) (Figure 4.11(b)). Each row x of this vertical panel is of size μ and must be replaced by xU^{-1} for a computation cost of $\frac{1}{2}\mu^2 w$.

   The most communication-efficient policy to implement this update is to keep the pivot matrix in place and to move around the rows of the vertical panel. Each row must be brought and sent back after update, for a total communication cost of $2\mu c$.

   At the k-th step, this update has then an overall communication cost of $2\mu(r - k\mu)c$ and an overall computation cost of $\frac{1}{2}\mu^2(r - k\mu)w$.

3. Update the μ rows at the right of the pivot matrix (horizontal panel) (Figure 4.11(c)). Each column y of this horizontal panel is of size μ and must be replaced by $L^{-1}y$ for a computation cost of $\frac{1}{2}\mu^2 w$.

Figure 4.10: Scheme for LU factorization at step k.



(a) The pivot matrix is factored.

(b) Update of vertical panel. A row x is replaced by xU^{-1}.

(c) Update of horizontal panel. A column y is replaced by $L^{-1}y$.

(d) μ columns of the core matrix are updated using the vertical panel and μ columns of the horizontal panel.

This case is symmetrical to the previous one. Therefore, we follow the same policy and at the k-th step, this update has an overall communication cost of $2\mu(r-k\mu)c$ and an overall computation cost of $\frac{1}{2}\mu^2(r-k\mu)w$.

4. Update the core matrix (square matrix of the last $(r-k\mu)$ rows and columns) (Figure 4.11(d)). This is a rank-μ update. Contrary to matrix multiplication, the most communication-efficient policy is to not keep the result matrix in memory, but either a $\mu \times \mu$ square block of the vertical panel or of the horizontal panel (both solutions are symmetrical). Arbitrarily, we then decide to keep in memory a chunk of the horizontal panel. Then to update a row vector x of the core matrix, we need to bring to that vector the corresponding row of the vertical panel, and then to send back the updated value of x. This has a communication cost of $3\mu c$ and a computation cost of μ^2.

   At the k-th step, this update for μ columns of the core matrix has an overall communication cost of $(\mu^2 + 3(r-k\mu)\mu)c$ (counting the communications necessary to initially bring the μ^2 elements of the horizontal panel) and an overall computation cost of $(r-k\mu)\mu^2 w$.

   Therefore, at the k-th step, this update has an overall communication cost of $(\frac{r}{\mu}-k)(\mu^2 + 3(r-k\mu)\mu)c$ and an overall computation cost of $(\frac{r}{\mu}-k)(r-k\mu)\mu^2 w$.

Using the above scheme, the overall communication cost of the LU factorization is

$$\sum_{k=1}^{\frac{r}{\mu}} \left(2\mu^2 + 4\mu(r-k\mu) + \left(\frac{r}{\mu}-k\right)(\mu^2 + 3(r-k\mu)\mu)\right) c = \left(\frac{r^3}{\mu} - r^2 + 2\mu r\right) c,$$

while the overall computation cost is

$$\sum_{k=1}^{\frac{r}{\mu}} \left(\mu^3 + \mu^2(r-k\mu) + \left(\frac{r}{\mu}-k\right)(r-k\mu)\mu^2\right) w = \frac{1}{3}\left(r^3 + 2\mu^2 r\right) w.$$

4.7.2 Algorithm for homogeneous clusters

The most time-consuming part of the factorization is the update of the core matrix, as it has an overall cost of $\left(\frac{1}{3}r^3 - \frac{1}{2}\mu r^2 + \frac{1}{6}\mu^2 r\right) w$. Therefore, we want to parallelize this update by allocating blocks of μ columns of the core matrix to different processors. Just as for matrix multiplication, we would like to determine the optimal number of participating workers $\mathfrak{P}$. For that purpose, we proceed as previously. On the communication side, we know that in a round (each worker updating μ columns entirely), the master sends to each worker μ^2 blocks of the horizontal panel, then sends to each worker the $\mu(r - k\mu)$ blocks of the vertical panel, and exchanges with each of them $2\mu(r - k\mu)$ blocks of the core matrix ($\mu(r - k\mu)$ received and later sent back after update). Also during this round, on the computation side, each worker computes $\mu^2(r - k\mu)$ block updates.

If we enroll too many processors, the communication capacity of the master will be exceeded. There is a limit on the number of blocks sent per time unit, hence on the maximal processor number $\mathfrak{P}$, which we compute as follows: $\mathfrak{P}$ is the smallest integer such that

$$(\mu^2 + 3\mu(r - k\mu))c\mathfrak{P} \geq \mu^2(r - k\mu)w.$$

We obtain that

$$\mathfrak{P} = \left\lceil \frac{\mu w}{3c} \right\rceil,$$

while neglecting the term μ^2 in the communication cost, as we assume $\frac{r}{\mu}$ to be large.

Once the resource selection is performed, we propose a straightforward algorithm: a single processor is responsible for the factorization of the pivot matrix and of the update of the vertical and horizontal panels, and then $\mathfrak{P}$ processors work in parallel at the update of the core matrix.

4.7.3 Algorithm for heterogeneous platforms

In this section, we simply sketch the algorithm for heterogeneous platforms. When targeting heterogeneous platforms, there is a big difference between LU factorization and matrix multiplication. Indeed, for LU once the size μ of the pivot matrix is fixed, all processors have to deal with it, whatever their memory capacities. There was no such fixed common constant for matrix multiplication. Therefore, a crucial step for heterogeneous platforms is to determine the size μ of the pivot matrix. Note that two pivot matrices at two different steps of the factorization may have different sizes, the constraint is that all workers must use the same size at any given step of the elimination.

In theory, the memory size of the workers can be arbitrary. In practice however, memory size usually is an integral number of Gigabytes, and at most a few tens of Gigabytes. So it is feasible to exhaustively study all the possible values of μ, estimate the processing time for each value, and then pick the best one. Therefore, in the following we assume the value of μ has been chosen, i.e., the pivot matrix is of a known size $\mu \times \mu$.

The memory layout used by each worker P_i follows the same policy than for the homogeneous case:

- a chunk of the horizontal panel is kept in memory,
- rows of the horizontal panel are sent to P_i,
- and rows of the core matrix are sent to P_i and are returned to the master after update.

If $\mu_i = \mu$, processor P_i operates exactly as for the homogeneous case. But if the memory capacity of P_i does not perfectly correspond to the size chosen for the pivot matrix, we still have to decide the shape of the chunk of the horizontal panel that processor P_i is going to keep in its memory. We have two cases to consider:

1. $\mu_i < \mu$. In other words, P_i has not enough memory. Then we can imagine two different shapes for the horizontal panel chunk:

   (a) Square chunk, i.e., the chunk is of size $\mu_i \times \mu_i$. Then, for each update the master must send to P_i a row of size μ_i of the horizontal panel and a row of size μ_i of the core matrix, and P_i sends back after update the row of the core matrix. Hence a communication cost of $3\mu_i c$ for μ_i^2 computations. The computation-to-communication cost induced by this chunk shape is then:

   $$\frac{\mu_i^2 w}{3\mu_i c} = \frac{\mu_i w}{3c}.$$

   (b) Set of whole columns of the horizontal panel, i.e., the chunk is of size $\mu \times \left(\frac{\mu_i^2}{\mu}\right)$. Then, for each update the master must send to P_i a row of size μ of the horizontal panel and a row of size $\frac{\mu_i^2}{\mu}$ of the core matrix, and P_i sends back after update the row of the core matrix. Hence a communication cost of $\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c$ for μ_i^2 computations. The computation to communication cost induced by this chunk shape is then:

   $$\frac{\mu_i^2 w}{\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c}.$$

   The choice of the policy depends on the ratio $\frac{\mu_i}{\mu}$. Indeed,

   $$\begin{aligned} \frac{\mu_i^2 w}{3\mu_i c} < \frac{\mu_i^2 w}{\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c} &\Leftrightarrow \left(\mu + 2\frac{\mu_i^2}{\mu}\right) c < 3\mu_i c \\ &\Leftrightarrow \left(2\frac{\mu_i}{\mu} - 1\right)\left(\frac{\mu_i}{\mu} - 1\right) < 0 \end{aligned}$$

   Therefore, the square chunk approach is more efficient if and only if $\mu_i \leq \frac{1}{2}\mu$.

2. $\mu_i > \mu$. In other words, P_i has more memory than necessary to hold a square matrix like the pivot matrix, that is a matrix of size $\mu \times \mu$. In that case, we propose to divide the memory of P_i into $\left\lfloor \frac{\mu_i^2}{\mu^2} \right\rfloor$ square chunks of size μ, and to use this processor as if there were in fact $\left\lfloor \frac{\mu_i^2}{\mu^2} \right\rfloor$ processors with a memory of size μ^2.

So far, we have assumed we knew the value of μ and we have proposed memory layout for the workers. We still have to decide which processor to enroll in the computation. We perform the resource selection as for matrix multiplication: we decide to assign only full matrix column blocks of the core matrix and of the horizontal panel to workers, and we actually perform resource selection using the same selection algorithms than for matrix-multiplication.

The overall process to define a solution is then:

1. For each possible value of μ do
   (a) Find the processor which will be the fastest to factor the pivot matrix, and to update the horizontal and vertical panels.
   (b) Perform resource selection and then estimate the running time of the update of the core-matrix.
2. Retain the solution leading to the best (estimated) overall running time.

4.8 MPI experiments

In this section, we aim at validating the previous theoretical results and algorithms. We conduct a variety of MPI experiments to compare our new schemes with several algorithms from the literature. We first restrict our study to homogeneous platforms. Even in this simpler framework, using a sophisticated memory management turns out to be very important. For heterogeneous platforms, we assess the impact of the degree of heterogeneity (in processor speed, link bandwidth, and memory capacity) on the performance of the various algorithms.

We start with a description of the platforms (Section 4.8.1), and of the algorithms (Section 4.8.2). Then we describe the experiments, justify their purpose, and discuss the results, first for homogeneous platforms (Section 4.8.3), then for heterogeneous platforms (Section 4.8.4).

The code and the experimental results can be downloaded from:

http://graal.ens-lyon.fr/~jfpineau/Downloads/matrix_product.tgz.

4.8.1 Platforms

For our homogeneous experiments we used a cluster of 64 Xeon 3.2GHz dual-processor nodes, located at the University of Tennessee in Knoxville. Each cluster node runs the Linux operating system and has 4 GB of memory, but we only use 512 MB of memory to further stress the impact of limited memories. The nodes are connected with a switched 100 Mbps Fast Ethernet network. In order to build a master-worker platform, we arbitrarily choose one processor as the master, and the other ones as workers. Finally we used *MPI_WTime* as timer in all experiments.

For our heterogeneous experiments we used a heterogeneous cluster composed of twenty-seven processors located at the École Normale Supérieure of Lyon. It is composed of four different homogeneous sets of machines. The different sets are:

- 8 SuperMicro servers 5013-GM, with processors P4 2.4 GHz;
- 5 SuperMicro servers 6013PI, with processors P4 Xeon 2.4 GHz;
- 7 SuperMicro servers 5013SI, with processors P4 Xeon 2.6 GHz;
- 7 SuperMicro servers IDE250W, with processors P4 2.8 GHz.

All nodes have 1 GB of memory and are running the Linux operating system. The nodes are connected with a switched 10 Mbps Fast Ethernet network.

Like in Chapter 3, this platform may not be as heterogeneous as we would like, so we sometimes artificially modify its heterogeneity. In order to artificially slow down a communication link, we send the same message several times to one worker. The same idea works for processor speeds: we ask a worker to compute a given matrix-product several times in order to slow down its computation capability. One can note that all MPI experiments implement the one-port model.

In all experiments, except the last batch, we used nine processors: one master and eight workers. We restricted the number of workers to eight after an initial experiment using the whole platform. This experiment showed that, because of platform parameters and of memory limitations, if the master serves five processors or more, on average one processor is idle at any time. In all experiments we compare the execution time needed by the algorithms using our memory allocation to the execution time of the other algorithms. We also point out the number of processors used by each algorithm, an important parameter when comparing execution times.

4.8.2 Algorithms

The algorithms we developed in section 4.5 and 4.6 define both a memory allocation and a resource selection. During the experiments, we want to test both.

We choose four different algorithms from the literature. The closest work addressing our problem is Toledo's out-of-core algorithm [140]. Hence, this work will serve as the baseline reference, and will be used to compare the performance of our memory allocation. In order to test the performance of our resource selection, we will launch hybrid algorithms, i.e., algorithms which use our memory layout and are based on classical principles such as round-robin, min-min [104], or a dynamic demand-driven approach. They will use our memory layout to divide matrices into chunks and to determine in which order chunks have to be sent to participating workers, but these participating workers and the order in which the master serves them will differ.

To summarize, here are the description of all algorithms launched during the MPI experiments. First, our algorithms:

Homogeneous algorithm (**Hom**) is our homogeneous algorithm. It makes resource selection and sends blocks to the selected workers in a round-robin fashion. When run on a heterogeneous platform, it tries to build a very simple homogeneous platform. As the algorithm's only constraint is to send same size blocks to all participating workers, for a given memory size, we consider the homogeneous virtual platform composed of those workers having at least that amount of memory, and we estimate the total execution time of our homogeneous algorithm, for the targeted matrix-product, on that virtual platform (the apparent processor speed is the minimum of the processor speeds, the apparent communication bandwidth is the minimum of the communication bandwidths). We do this process for all the different memory sizes present in the actual platform, and we pick the virtual platform that minimizes the total estimated execution time.

Homogeneous algorithm, Improved (**HomI**) is our homogeneous algorithm running on a more carefully chosen homogeneous platform. For each memory size, communication speed, and computation speed present in an heterogeneous platform, we consider the homogeneous virtual platform composed of those workers having at least that performance. Then, we compute the total execution time of our homogeneous algorithm, for the targeted matrix-product, on that virtual platform (the apparent processor speed is the current processor speed, the apparent communication bandwidth is the current communication bandwidth). We do this process for all the existing values, and we pick the virtual platform that minimizes the total execution time.

Heterogeneous algorithm (**Het**) is our heterogeneous algorithm, described in Section 4.6.

The other algorithms use our memory layout, and can overlapp one update of $\mathcal{C}$ and the reception of the next column of $\mathcal{A}$ and the next row of $\mathcal{B}$:

Overlapped Round-Robin, Optimized Memory Layout (**ORROML**) sends tasks to all available workers in a round-robin fashion. It does not make any resource selection.

Overlapped Min-Min, Optimized Memory Layout (**OMMOML**) is a static scheduling heuristic, which sends the next block to the first worker that will finish it. As it is looking for potential workers in a given order, this algorithm performs some resource selection too. Theoretically, as our homogeneous resource selection ensures that the first worker is free to compute when we finish to send blocks to the others, **OMMOML** and **Hom** should have a similar behavior on homogeneous platforms.

Overlapped Demand-Driven, Optimized Memory Layout (**ODDOML**) is a demand-driven algorithm. In our memory layout, two buffers of size μ_i are reserved for matrix $\mathcal{A}$, and two for matrix $\mathcal{B}$. In order to use the two available extra buffers (the second for $\mathcal{A}$ and the second for $\mathcal{B}$), one sends the next block to the first worker which can receive it. This would be a dynamic version of our algorithm, if it took worker selection into account.

Then, we have Toledo's:

Block Matrix Multiply (**BMM**) is Toledo's algorithm [140]. It splits each worker memory equally into three parts, and allocates one slot for a square block of $\mathcal{A}$, another for a square block of $\mathcal{B}$, and the last one for a square block of $\mathcal{C}$, with the square block having the same size. It sends blocks to the workers in a demand-driven fashion, when a worker is free for computation. First a worker receives a block of $\mathcal{C}$, then it receives corresponding blocks of $\mathcal{A}$ and $\mathcal{B}$ in order to update $\mathcal{C}$, until $\mathcal{C}$ is fully computed.

The last two remaining algorithms are variants of the previous ones. We only test them on homogeneous platforms because of their bad performance:

Demand-Driven, Optimized Memory Layout (**DDOML**) is another demand-driven algorithm, close to **ODDOML**. It sends the next block to the first worker which is free for computation. As workers will never have to receive and compute at the same time, the algorithm has no extra buffer, so the memory available to store $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ is slightly greater. This may change the value of the μ_i's and so the behavior of the algorithm.

Overlapped Block Matrix Multiply (**OBMM**) is our attempt to improve the Toledo's algorithm. We try to overlap the communications and the computations of the workers. To that purpose, we split each worker memory into five parts, to receive one block of $\mathcal{A}$ and one block of $\mathcal{B}$ while the previously received blocks are used to update $\mathcal{C}$.

Note that all algorithms using our optimized memory layout are considering matrices as composed of square blocks of size $q \times q = 80 \times 80$, while **BMM** loads three panels, each of size one third of the available memory, for $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.

ODDOML will be particularly interesting to analyze, because it uses our memory allocation, but has the same resource selection than Toledo's. It is then the perfect algorithm to see the impact of our resource selection (by comparing it with **Het**) and our memory allocation (by comparison with **BMM**).

When launching an algorithm on the platform, the very first step we do is to determine the platform's parameters. For that purpose, we launch a benchmark on it, in order to get the memory size, the communication speed, and the computation speed. The different speeds are determined by sending and computing a square block of size $q \times q$ ten times on each worker, and

computing the median of the times obtained. This step takes between twenty and eighty seconds, depending on the speed of the workers, and is made before each algorithm, even **ORROML**, **ODDOML** and **BMM**, which only need the memory size. This step represents at most two percent of the total time of execution.

In the following section, the times given takes into account the decision process of the algorithms, i.e., the simulation of the eight different versions of the resource selection for **Het**, the construction of an homogeneous platform for **Hom** and **HomI**, etc.

4.8.3 Experiments on homogeneous platforms

In this section we validate the superiority of our homogeneous algorithm on homogeneous platforms. We consider all algorithms except the heterogeneous one, and **HomI**. We have built several experimental protocols in order to assess the performance of the various algorithms.

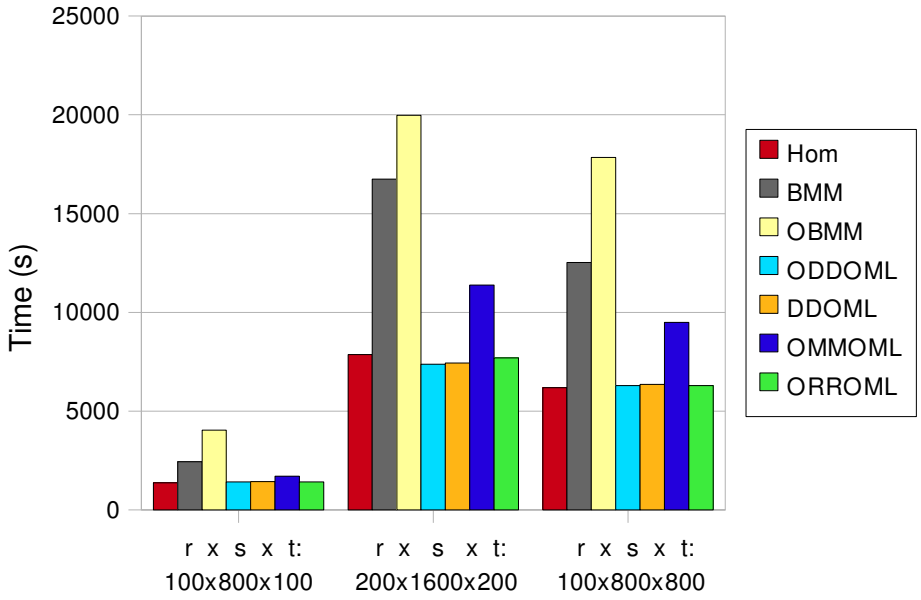


Figure 4.11: Performance of the algorithms on different matrices.

Experiments with different matrix sizes

In the first set of experiments, we test the different algorithms on matrices of different sizes and shapes. The matrices we are multiplying are of actual size

1. 8000×8000 for $\mathcal{A}$ and 8000×64000 for $\mathcal{B}$;
2. 16000×16000 for $\mathcal{A}$ and 16000×128000 for $\mathcal{B}$;
3. 8000×64000 for $\mathcal{A}$ and 64000×64000 for $\mathcal{B}$.

Figure 4.11 presents the results of this first set of experiments. We first remark that the shape of the results of the three experiments is the same for all matrix sizes. We also underline the superiority of most of the algorithms which use our memory allocation against **BMM**: **Hom**,

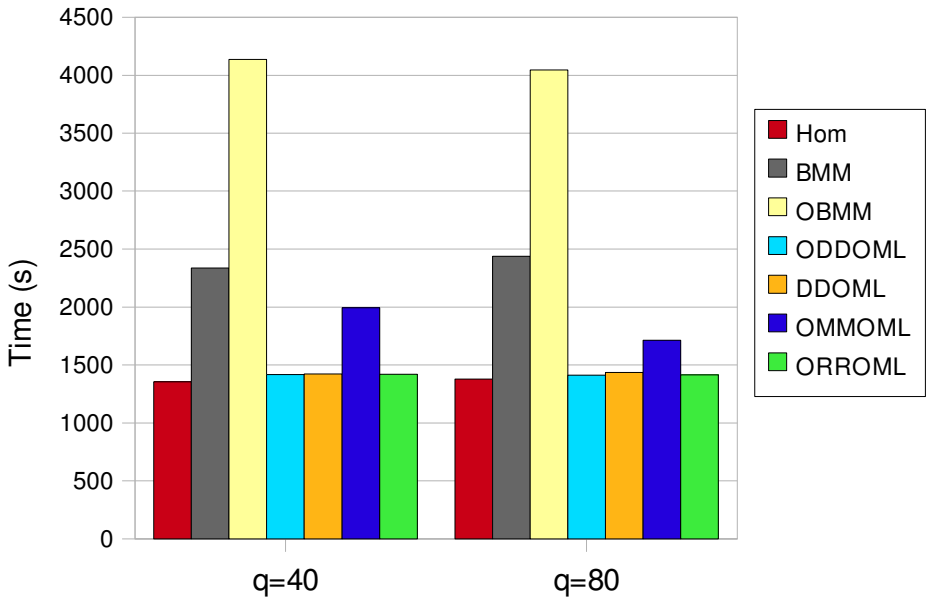


Figure 4.12: Impact of block size q on algorithm performance.

ORROML, **ODDOML**, and **DDOML** are the best algorithms and have similar performance. Only **OMMOML** needs more time to complete its execution. This delay comes from its resource selection: it only uses three workers. For instance, **Hom** uses four workers, and is as competitive as the other algorithms which all use the eight available workers. This difference can be explained by the resource selection process. The times of communications and computations are computed before the scheduling by sending a task to each slave. Then **OMMOML** will use those values, which may not be as accurate as needed, whereas **Hom** will use the worst time for communications and computations in order to consider a homogeneous platform.

Impact of the block size q

In the second set of experiments we check whether the choice of q was wise. For that purpose, we launch the algorithms on matrices of size 8000×8000 and 8000×64000, changing from one experiment to another the size of the elementary square blocks. Then q will be respectively equal to 40 and 80. As the global matrix size is the same in both experiments, we expect both results to be the same.

The results are displayed in Figure 4.12. **BMM** and **OBMM** have same execution times in both experiments as these algorithms do not split matrices into elementary square blocks of size $q \times q$ but, instead, call the Level 3 BLAS routines directly on the whole $\sqrt{\frac{m}{3}} \times \sqrt{\frac{m}{3}}$ matrices. In the two cases we see that the time of the algorithms are similar. We point out that this experiment shows that the choice of q has little impact on the algorithms' performance.

Impact of the memory size of workers

In the third set of experiments we investigate the impact of the worker memory size onto the performance of the algorithms. In order to have reasonable execution times, we use matrices of size 16000×16000 and 16000×64000, and the memory size will be either 132MB or 512MB.

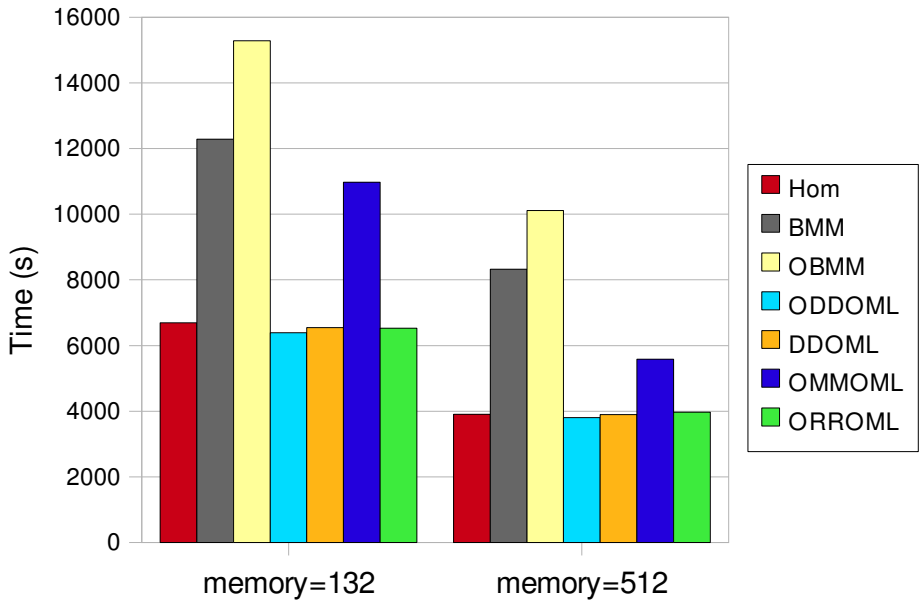


Figure 4.13: Impact of memory size on algorithm performance.

We choose these values to reduce side effects due to the partition of the matrices into blocks of size $\mu q \times \mu q$.

Figure 4.13 presents the experimental results. As expected, the performance increases with the amount of memory available. It is interesting to underline that our resource selection is efficient. **Hom** will use respectively two and four workers when the available memory increases, compared to the other algorithms which will use all eight available workers on each test. **OMMOML** also makes some resource selection, but performs worse.

Stability of execution times

In the fourth and last set of those experiments we check the stability of the previous results. To that purpose we launch the same execution five times, in order to determine the maximum gap between two runs.

Figure 4.14 shows that the maximum difference between two runs of the same experiment is around 6%. Thus if two algorithms have no more than 6% of difference in execution times, they should be considered as similar.

Conclusion

These experiments stress the superiority of our memory allocation on homogeneous platforms. Furthermore, our homogeneous algorithm is as competitive as the others but uses fewer resources.

We are now going to consider heterogeneous platforms. As **OBMM** always has significantly worse performance than **BMM**, we will no longer consider this algorithm. As algorithms **DDOML** and **ODDOML** are very close and have comparable execution times, the latter having slightly (but not significantly) better performance, there is no need to consider both, and we drop the former from our study.

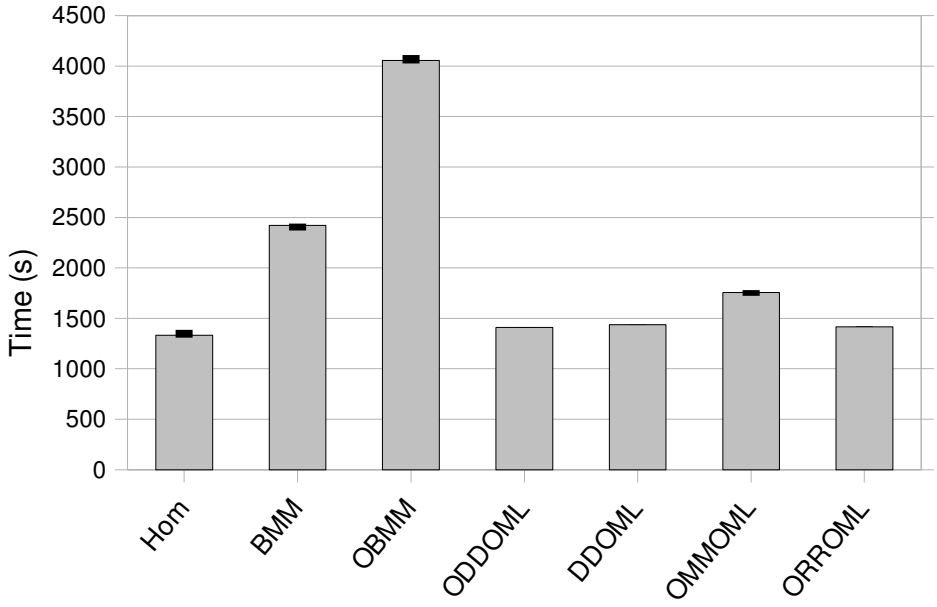


Figure 4.14: Variation of algorithm execution times.

4.8.4 Experiments on heterogeneous platforms

In this section we compare the algorithms on heterogeneous platforms. All the algorithms using our optimized memory layout are still considering matrices as composed of square blocks of size $q \times q = 80 \times 80$.

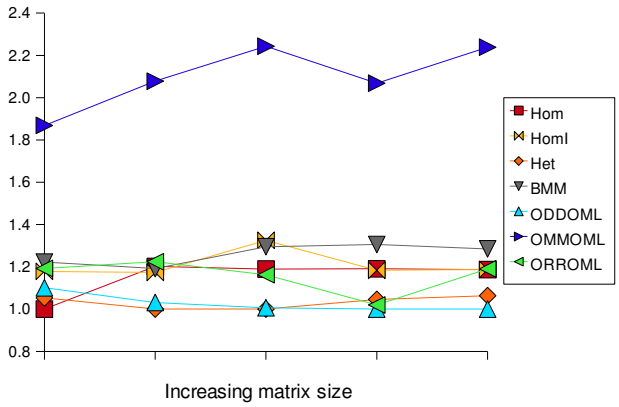
In the first three sets of experiments, we only have one parameter of heterogeneity, either the amount of memory, the communication speed, or the computation speed. We test the algorithms on such platforms with five matrices of increasing size. As we do not want to change several parameters at a time, we only change the value of parameter s (rather than, for instance, always consider square matrices). The matrix $\mathcal{A}$ is of size 8000×8000 whereas $\mathcal{B}$ is of increasing sizes 8000×64000, 8000×80000, 8000×96000, 8000×112000, and 8000×128000. For all other experiments, $\mathcal{A}$ is of size 8000×8000 and $\mathcal{B}$ is of size 8000×80000.

The heterogeneous workers have different memory capacities, which implies that each algorithm, even **BMM**, assigns them chunks of different sizes. In order to simplify the global partitioning of matrix $\mathcal{C}$, we decide to only assign workers full matrix column blocks.

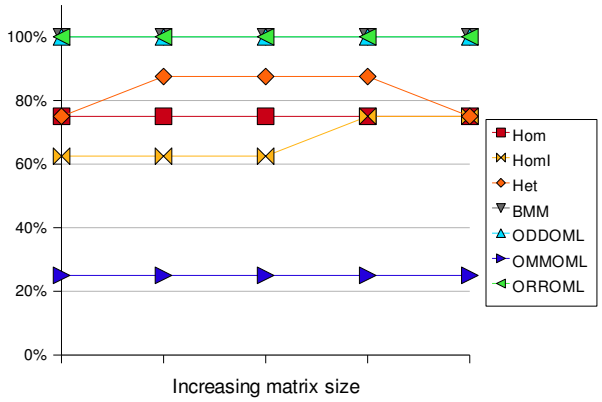
As we want to assess whether the performance of any studied algorithm depends on the matrix size, we look at the *relative cost* of the algorithms rather that at their absolute execution times (absolute execution times are listed in Appendix B). The relative cost of a given algorithm on a particular instance is equal to the *makespan* achieved on that instance by the algorithm, divided by the minimum *makespan* achieved on that instance. Using relative costs also enables us to build statistics on the performance of algorithms.

Beside makespan or relative cost, we take into account a second parameter: the number of processors used. To assess the efficiency of a given algorithm, we look at the utilization of the platform, i.e., the percentage of enrolled processors.

Figure 4.15: Heterogeneous memory.



(a) Cost of algorithms.



(b) Platform's utilization.

Heterogeneous memory sizes

In the first set of experiments we assess the cost of our algorithms with respect to memory heterogeneity. We launch the algorithms on a homogeneous platform in terms of communication and computation capabilities, but where workers have different memory capacities. We suppose that two workers only have 256 MB of memory, four of them 512 MB, and the last two ones 1024 MB.

Figure 4.16(a) presents the relative cost of the algorithms, whose general shape is very similar for all five matrix sizes. **ODDOML** and our heterogeneous algorithm **Het** have the best *makespans*. At the other end of the spectrum, **OMMOML** is twice worse. In between, **Hom**, **HomI**, **ORROML**, and **BMM** are roughly twenty percent slower. To give an idea of the execution times, we report that **Het** needs about 2000 seconds to compute the product of the smallest matrices, and about 3500 seconds for the largest.

The variations in the cost of **BMM** can easily be justified. The memory layout used in

BMM is different from the other algorithms. Therefore the size of the matrix chunks used by **BMM** are different. The matrices are rather small (to be able to evaluate a significant number of algorithms on a significant number of platforms). Hence we observe some non negligible side effects (matrix size divided by chunk size not being a multiple of number of processors used). Therefore, for a given platform, some memory sizes are more favorable for **BMM** than for the algorithms using our memory layout. We will see throughout our experiments on heterogeneous platforms that even if sometimes these side effects help **BMM** achieve reasonable cost, they do not prevent it to sometimes achieve very bad cost.

If we look at the utilization of the platform (Figure 4.16(b)), one can remark that **OMMOML** only uses two workers, and is thus very thrifty, at the expense of its absolute cost. **Hom** is performing some resource selection contrarily to **ODDOML** which always uses all the processors. One can note that **HomI** is not better than **Hom**, despite its improved platform selection.

Heterogeneous communication links

In the second set of experiments, we assess the cost of our algorithms when communication links have heterogeneous capabilities. The target platform is composed of two workers with a 10Mbps communication link, four workers with a 5Mbps communication link, and the last two ones have a 1Mbps communication link.

Figure 4.17(a) shows the relative cost. The superiority of our heterogeneous algorithm over **BMM** is clear.

Het, **HomI**, and **OMMOML** have excellent makespans, and make a good resource selection, as seen in Figure 4.17(b)). The first figure also shows the gap between **HomI** and **Hom**: **Hom** performs close to **ODDOML**, while **HomI** achieves a close to best makespan. This figure underlines the importance of carefully choosing the processors on which launching the algorithm: **Hom** only uses two processors because of the platform parameters and the way it extracts a homogeneous platform.

BMM has the worst makespan, which is 70 to 90 percent worse than the best one, and makes no resource selection, On the contrary, our heterogeneous algorithm, as well as **HomI** make excellent resource selection.

Concerning execution times, **Het** needs about 2500 seconds to compute the product of the smallest matrices, and about 5000 seconds for the largest.

Heterogeneous computation capabilities

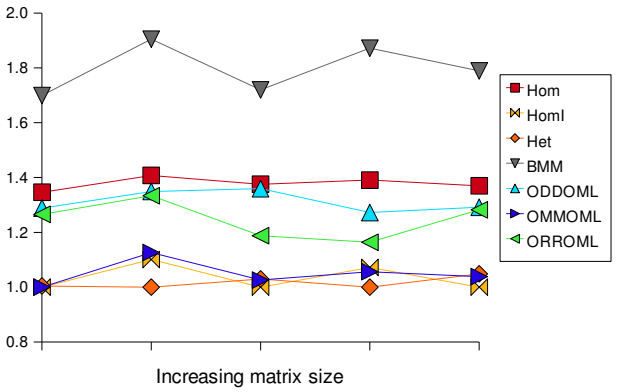
In the third set of experiments, we assess the cost of our algorithms when computation capabilities are heterogeneous. Workers have homogeneous communications and memory capacities, but different computation speeds. The platform is composed of two fast workers of speed s, four workers of speed $s/2$ and two workers of speed $s/4$.

In Figure 4.18(a), we see the relative cost obtained during this set of experiments. **BMM** performs well, but its makespan is larger than that of **Het**. Among the other algorithms, we also see that **ODDOML** has good performance.

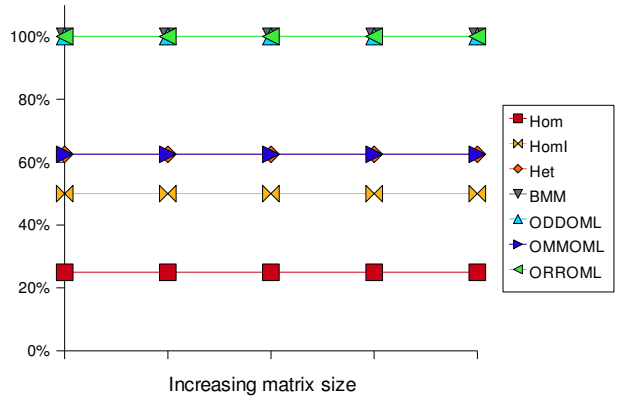
Moreover, looking at the platform's utilization in Figure 4.18(b), we see that our algorithms enroll fewer resources during execution. One can also remark that **Het** uses more and more processors as matrix size increases.

During these experiments, **Het** needs about 2000 seconds to compute the product of the smallest matrices, and about 4000 seconds for the largest one.

Figure 4.16: Heterogeneous communication links.



(a) Cost of algorithms.



(b) Platform's utilization.

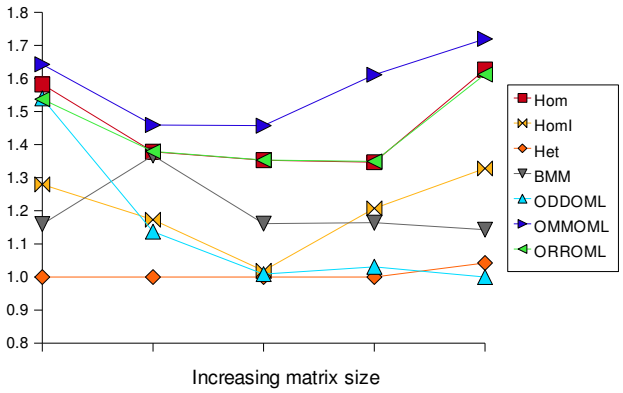
Platforms with two homogeneous subsets

In the fourth set of experiments, we split the platform into two homogeneous sets containing four workers each.

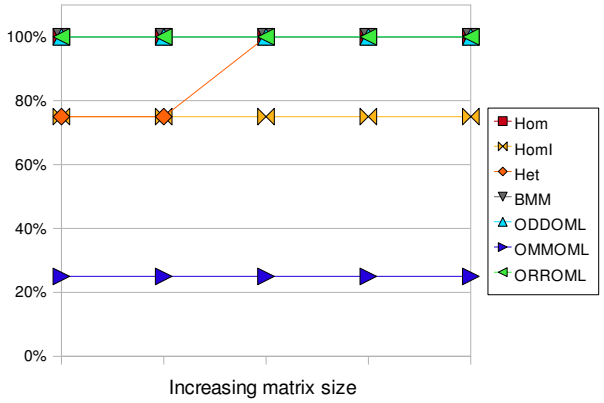
In the first setting, we have four workers with fast computation speeds but slow communication links, and four workers with slow computation speed but fast communication links. In the first experiment (Figure 4.19(a)), the ratio between slow and fast communication speeds is equal to the ratio between slow and fast computation speeds, and is set to two. In the second experiment (Figure 4.19(b)), both ratios are set to four.

Figure 4.19(a) and Figure 4.19(b) are similar, as the two platforms have the same shape. On the first platform, **Het** and **ODDOML** have the best makespan. In Figure 4.19(b), we see that **ODDOML** slightly outperforms our heterogeneous algorithm (by roughly 12%). However, looking at the platform's utilization, we remark that the makespan achieved by **Het** is in fact obtained with two fewer workers than for **ODDOML**.

Figure 4.17: Heterogeneous computations.



(a) Cost of algorithms.

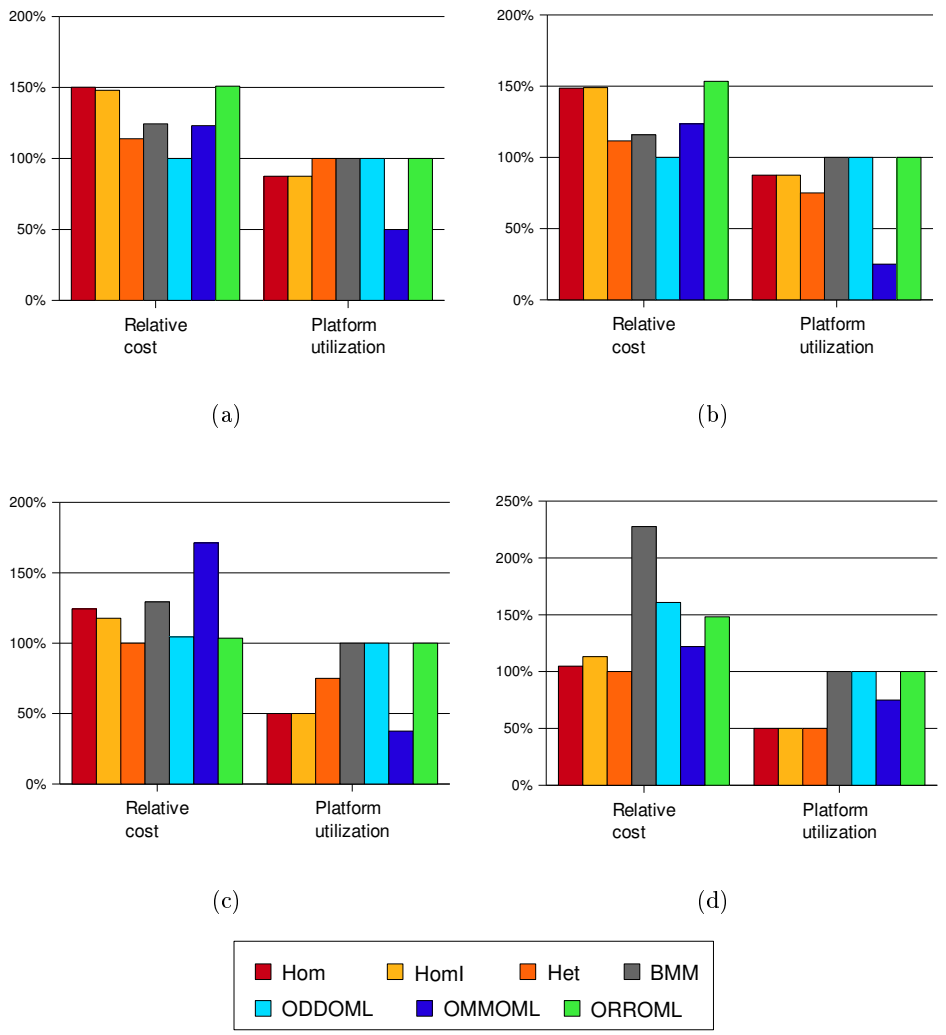


(b) Platform's utilization.

In the second setting, one half of the processors have slow communication links and small memories, and the other half has both fast communication links and large memories. In the first experiment (Figure 4.19(c)), the ratio between slow and fast communication speeds is equal to the ration between small and large memories, and is set to two. In the second experiment (Figure 4.19(d)), both ratios are set to four. On these platforms, our homogeneous algorithms extract an excellent homogeneous platform, composed of the four powerful workers, i.e., with large memories and fast communication links, and achieve a good makespan. Our heterogeneous algorithm achieves the best makespan, but need two more workers than **Hom**.

The second experiment (Figure 4.19(d)) having more drastic characteristics, its conclusions are more obvious. All algorithms that do not perform resource selections achieve bad relative performance as they enroll inefficient processors. This is particularly true for **BMM** and **ODDOML**.

Figure 4.18: Platforms with two homogeneous subsets.

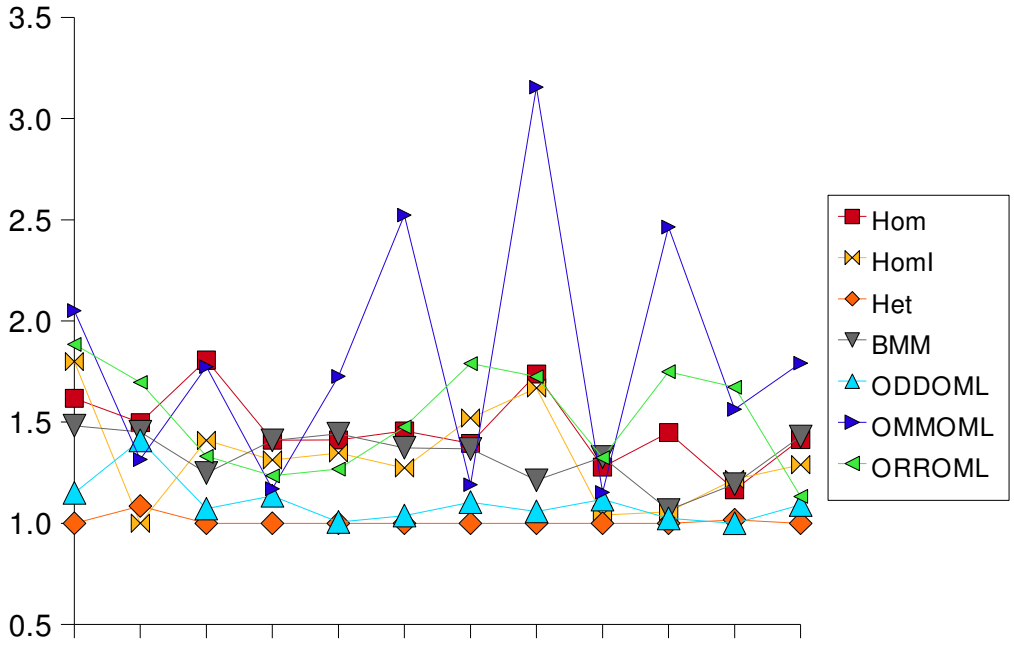


Fully heterogeneous platforms

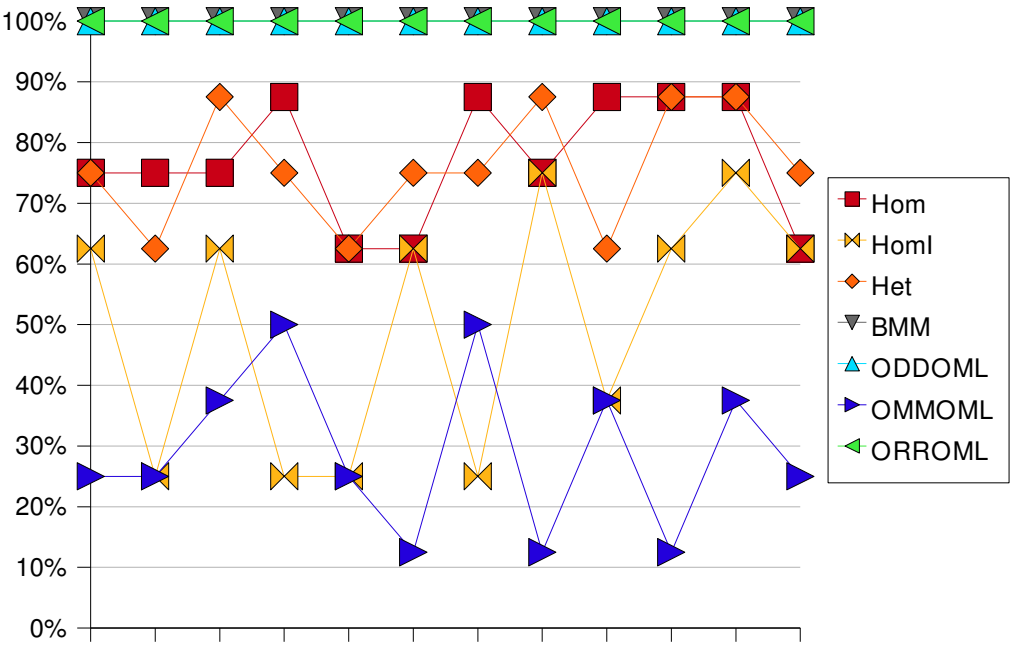
In the fifth set of experiments we have a fully heterogeneous platform. Communication links, computation capabilities, and memory capacities can take two different values, which leads to eight possibilities, one per worker. We build that way two different platforms by fixing the ratio between the small and large values for each characteristics to 2 in the first setting and 4 in the second one (first two columns in Figures 4.20(a) and 4.20(b)). In order to show that our heterogeneous algorithm works on any heterogeneous platform, we also randomly create ten different platforms (last ten columns in the same figures). The ratio between minimum and maximum values of communication links, computation capacities and memory size is up to four. Matrix $\mathcal{A}$ is of size 8000×8000 and $\mathcal{B}$ of size 8000×80000.

The results of these experiments are summarized in Figures 4.20(a) and 4.20(b). We see that **Het** achieves the best makespan for all but two of the 12 platforms, and in the remaining cases

Figure 4.19: Fully heterogeneous platforms.



(a) Cost of algorithms.



(b) Platform's utilization.

is never more than 9% and 2% away from the best studied algorithm. All the other algorithms are, at least once, more than 41% away from the best solution. For example, **ORROML** can be up to 88% worse than the best achieved makespan. Only **ODDOML** achieves reasonable makespan on average but it does not perform any resource selection.

The platform's utilization of **Het** is efficient, as it spares resources while achieving a good makespan. The other algorithms which make resources selection are our homogeneous algorithms (**Hom** and **HomI**), and the unusable **OMMOML**, whose makespan can be 215% away from the best solution. But even if our improved homogeneous algorithm performs a good resources selection, the makespan it achieves can be up to 80% larger than the best makespan, and is 34% larger on average.

Depending on the experiments, **Het** needs between 2700 seconds and 6000 seconds.

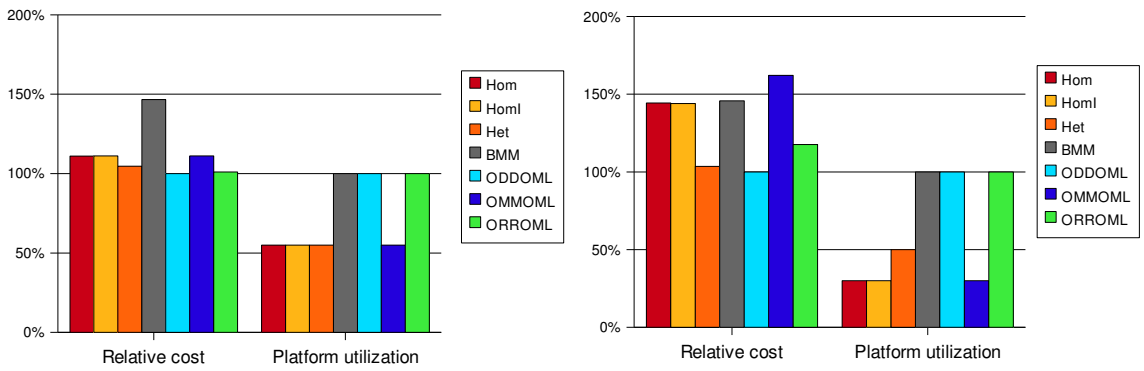
Real platform

In this set of experiments, we use almost all the processors of our platform. We do not modify the communication speeds nor the computation speeds of the workers. We take five processors of each of the four sets of machines, which gives us a rather homogeneous platform. We either use this platform "as is" (*August 2007 configuration* of Figure 4.21(a)) or we limit the amount of memory available on each processor to its value before the last memory upgrade (*November 2006 configuration* of Figure 4.21(b)). The actual platform was then:

- 5 SuperMicro servers 5013-GM, with processors P4 2.4 GHz with 256 MB of memory;
- 5 SuperMicro servers 6013PI, with processors P4 Xeon 2.4 GHz with 1 GB of memory;
- 5 SuperMicro servers 5013SI, with processors P4 Xeon 2.6 GHz with 1 GB of memory;
- 5 SuperMicro servers IDE250W, with processors P4 2.8 GHz with 256 MB of memory.

We use an extra processor as the master. The matrix are of size 8000×8000 for $\mathcal{A}$ and 8000×320000 for $\mathcal{B}$.

Figure 4.20: Real platform.



(a) Real memory

(b) Limited memory

The results of these experiments are summarized in Figure 4.20. The results on the actual platform are similar to those obtained on homogeneous platforms. All the algorithms but **BMM** have similar makespan. All algorithms making resource selection use eleven workers among the twenty available.

The results of the experiment on the older version of the platform are very similar to the ones previously obtained on memory heterogeneous platforms. **ODDOML** and our heterogeneous algorithm **Het** achieve the best *makespans*. Then we have **ORROML**, our homogeneous algorithms, **BMM**, and finally **OMMOML** which is 60% worse than **Het**. The execution time is around 7800 seconds for **Het**. If we look at resource selection, **Het** uses only the ten workers which have 1 GB of memory, and achieves a makespan close to **ODDOML**'s which uses the whole platform. On another side, **Hom**, **HomI**, and **OMMOML** use six workers with small memories. All other algorithms use the whole platform.

Summary

We summarize here all our MPI experiments. Figures 4.22(a) and 4.22(b) respectively present the relative cost and the platform's utilization obtained for each experiment by our heterogeneous algorithm (**Het**), by Toledo's algorithm (**BMM**), and the dynamic heuristic using our memory layout (**ODDOML**).

The results show the superiority of our memory allocation. Over the thirty-three experiments, **ODDOML** performs worse than **BMM** only once, on the computation heterogeneous platform, and with a small size matrix. This can be considered as a side effect of the split of the matrix, as for small size matrices it is better to split it into small square size blocks as **BMM** does.

Furthermore, if we add the resource selection of **Het**, not only we achieve, most of the time, the best makespan, but we achieve this makespan while sparing resources.

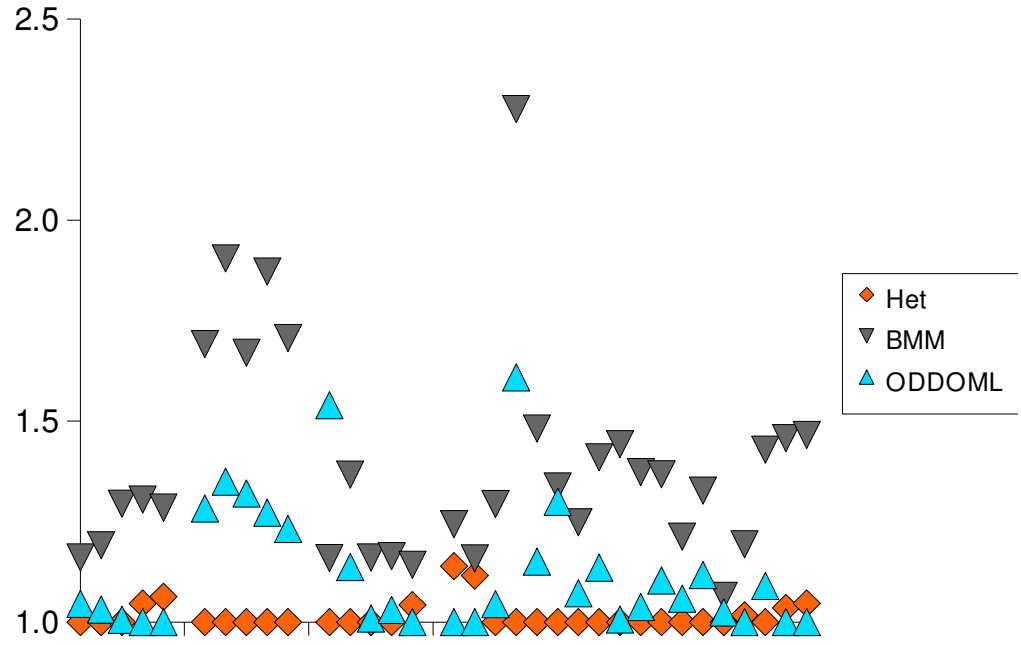
Finally, Table 4.3 displays the average and worst relative performance obtained over all experiments by these three algorithms. Using our memory layout (**ODDOML**) rather than Toledo's (**BMM**) enables us to gain 18% of execution time on average. When this is combined with resource selection, this enables us to gain additionally 11%, that is 27% against Toledo's running time. We achieve this significant gain while sparing 22% of the resources.

Our **Het** algorithm is on average 1% away from the best achieved makespan. At worst **Het** is 10% away from the best makespan, **ODDOML** 61%, and **BMM** 128%. Moreover, we have seen that 80% of the time, the cost of **Het** was in fact obtained thanks to a global resource selection.

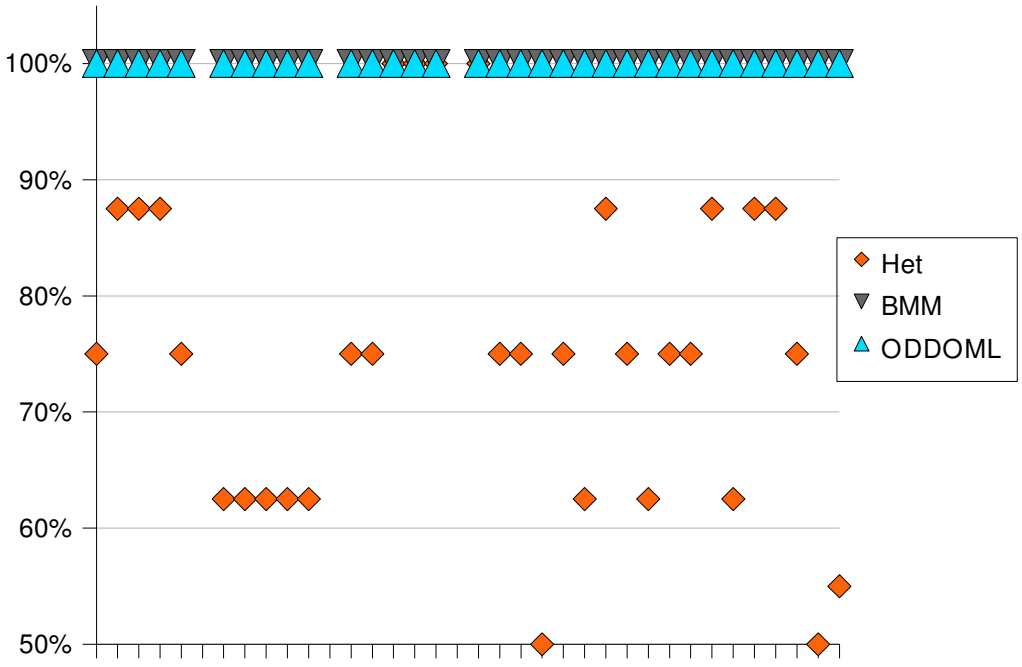
The steady-state approach described in Section 4.6.1 gives us an upper-bound on the best achievable throughput. This upper-bound is very optimistic as it assumes unbounded memories and does not take into account the communication costs due to the elements of matrix $\mathcal{C}$. This upper bound is nevertheless on average only 2.29 times greater than the throughput achieved by **Het** (and at worst is 3.42 times greater). Therefore, considering this upper-bound tells us that our **Het** algorithm not only has good relative cost when compared to the other algorithms, but also has very good absolute performance.

Altogether, we have thus been able to design an efficient, thrifty, and reliable algorithm.

Figure 4.21: Summary of experiments.



(a) Relative cost.



(b) Platform's utilization.

	Average cost	Worst cost
Het	1.01	1.10
BMM	1.39	2.28
ODDOML	1.13	1.61

Table 4.3: Average and worst relative cost.

4.9 Related work

In this section, we provide a brief overview of related papers, which we classify along the following five main lines:

Load balancing on heterogeneous platforms – Load balancing strategies for heterogeneous platforms have been widely studied. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. Some simple schedulers are available, but they use naive mapping strategies such as master-worker techniques or paradigms based upon the idea *"use the past to predict the future"*, i.e., use the currently observed speed of computation of each machine to decide for the next distribution of work [52, 53, 26]. Dynamic strategies such as *self-guided scheduling* [116] could be useful too. There is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration policies. Redundant computations might also be necessary to use a heterogeneous cluster at its best capabilities. However, dynamic strategies are outside the scope of this work (but mentioned here for the sake of completeness). Because we have a library designer's perspective, we concentrate on static allocation schemes that are less general and more difficult to design than dynamic approaches, but which are better suited for the implementation of fixed algorithms such as linear algebra kernels from the ScaLAPACK library [33].

Out-of-core linear algebra routines – As already mentioned, the design of parallel algorithms for limited memory processors is very similar to the design of out-of-core routines for classical parallel machines. On the theoretical side, Hong and Kung [80] investigate the I/O complexity of several computational kernels in their pioneering paper. Toledo [140] proposes a nice survey on the design of out-of-core algorithms for linear algebra, including dense and sparse computations. We refer to [140] for a complete list of implementations. The design principles followed by most implementations are introduced and analyzed by Dongarra et al. [61].

Matrix product on reconfigurable architectures– A similar thread of work, although in a different context, deals with reconfigurable architectures, either pipelined bus systems [101], or FPGAs [149]. In the latter approach, tradeoffs must be found to optimize the size of the on-chip memory and the available memory bandwidth, leading to partitioned algorithms that re-use data intensively.

Linear algebra algorithms on heterogeneous clusters – Several authors have dealt with the *static* implementation of matrix-multiplication algorithms on heterogeneous platforms. One simple approach is given by Kalinov and Lastovetsky [86]. Their idea is to achieve a perfect load-balance as follows: first they take a fixed layout of processors arranged as a collection of processor columns; then the load is evenly balanced *within* each processor

column independently; next the load is balanced *between* columns; this is the "heterogeneous block cyclic distribution" of [86]. Another approach is proposed by Crandall and Quinn [58], who propose a recursive partitioning algorithm, and by Kaddoura, Ranka, and Wang [85], who refine the latter algorithm and provide several variations. They report several numerical simulations. As pointed out in the introduction, theoretical results for matrix multiplication and LU decomposition on 2D-grids of heterogeneous processors are reported in [12], while extensions to general 2D partitioning are considered in [13]. See also Lastovetsky and Reddy [93] for another partitioning approach.

Recent papers aim at making easier the process of tuning linear algebra kernels on heterogeneous systems. Self-optimization methodologies are described by Cuenca et al. [59] and by Chen et al. [48]. Along the same line, Chakravarti et al. [45] describe an implementation of Cannon's algorithm using self-organizing agents on a peer-to-peer network.

Mixed approach for matrix multiplication – Some authors have shown that it is possible to automatically derive an efficient poly-algorithm to compute the product of two large dense square matrices. The selection of the right algorithm among all possible algorithms of the poly-algorithms is expressed as a combinatorial optimization problem. Nasri and Trystram have proposed a new parallel poly-algorithm which uses both advantages of standard and fast algorithms for multiplying two dense square matrices on conventional homogeneous clusters of PCs [107], while Jeddi and Nasri have proposed a mixed approach for matrix multiplication on both homogeneous and heterogeneous hierarchical clusters of SMPs [84].

4.10 Conclusion

The main contributions of this work are the following:

1. On the theoretical side, we have derived a new, tighter, bound on the minimal volume of communications needed to multiply two matrices. From this lower bound, we have defined an efficient memory layout, i.e., an algorithm to share among the three matrices the memory available on the workers.

2. On the practical side, starting from our memory layout, we have designed an algorithm for homogeneous platforms whose performance is quite close to the communication volume lower bound. We have extended this algorithm to deal with heterogeneous platforms, and discussed how to adapt the approach for LU factorization.

3. Through MPI experiments, we have shown that both our algorithms for homogeneous and heterogeneous platforms have far better performance than solutions using the memory layout proposed in [140]. Furthermore, this static homogeneous algorithm has similar performance than dynamic algorithms using the same memory layout, but uses fewer processors. It is therefore a very good candidate for deploying applications on regular, homogeneous platforms. Our heterogeneous algorithm also proves itself a very good candidate for deploying applications on heterogeneous platforms, providing better performance than any other solution, even the dynamic algorithms using the same memory layout. Furthermore, it uses fewer processors, and has a far better worst case.

As future work, it would be interesting to assess whether our memory layout could prove useful in the context of out-of-core algorithms executed on heterogeneous platforms.

If we change the model...

In this work, we supposed that the master had no processing capability. The main reason of this assumption was that if the master had processing capability, we could simulate it by an additional worker of same processing capability and a communication time equal to 0.

Unfortunately, this assumption is not always true, mainly because there is a strong hidden hypothesis. Making this reduction supposes that the master can be seen as any other worker, which it is not, because of its special role in the communication network, and the fact that it is the machine that holds all the data at the beginning.

Here, our goal is to perform the operation $\mathcal{C} = \mathcal{C} + \mathcal{A} \times \mathcal{B}$. If we suppose that the master has no processing capability (for example if the master is just a data server), then all updates of the matrix $\mathcal{C}$ have to be made on a worker. Which means that the master has to send the whole matrix $\mathcal{C}$. But if, instead of sending the old values of $\mathcal{C}$, the master only ask the slaves the results of $\mathcal{A} \times \mathcal{B}$, and performs itself the update of $\mathcal{C}$, then, depending of the communication to computation ratio, the results could be obtained faster. Of course, if we do not want any interference between the updates of $\mathcal{C}$ and the communications [91, 92], the master has to delay the next communication of elements of $\mathcal{A}$ and $\mathcal{B}$ while updating.

The key argument of our *maximum re-use* algorithm is that $\mathcal{C}$ is more important than $\mathcal{A}$ or $\mathcal{B}$, because it costs twice as many communications, and this argument vanishes under such a model. If we change our context, then there is a whole new problem that has to be studied. How the matrices have to be split? What matrix blocks should be reused? Those of $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$?

This small parenthesis has not being studied during this thesis, but could be a direction of future work.

Remark. *We would like to thank Jean-Yves L'Excellent for his remark about this special case.*

Chapter 5

Steady-State scheduling

We have shown in Chapter 3 that scheduling problems were difficult on heterogeneous clusters. In Chapter 4, however, we were able to design efficient algorithms for matrix multiplication, while not focusing on makespan minimization.

In this chapter, we will underline another method to bypass the difficulty of minimizing makespan: the steady-state approach.

5.1 Introduction

We are now dealing with the problem of scheduling collections of independent and identical tasks on a heterogeneous master-worker platform.

If the master-worker platform is homogeneous, i.e., if all workers have identical CPUs and same communication bandwidths to/from the master, then elementary greedy strategies, such as purely demand-driven approaches, will achieve an optimal throughput. On the contrary, if the platform gathers heterogeneous processors, connected to the master via different-speed links, then the previous strategies are likely to fail dramatically. This is because it is crucial to select which resources to enroll before initiating the work distribution [9, 114].

In a first part, we still target fully parallel applications, but we introduce a much more complex (and more realistic) framework than scheduling a single application. We envision a situation where users, or clients, submit several bag-of-tasks applications to a heterogeneous master-worker platform, using a classical client-server model. Applications are submitted on-line, which means that we do not know beforehand when applications will be submitted and what their characteristics will be. When several applications are executed simultaneously, they compete for hardware (network and CPU) resources. What is the scheduling objective in such a framework? A greedy approach would execute the applications sequentially in the order of their arrival, thereby optimizing the execution of each application onto the target platform. Such a simple approach is not likely to be satisfactory for the clients. For example, the greedy approach may delay the execution of the second application for a very long time, while it might have taken only a small fraction of the resources and few time-steps to execute it concurrently with the first one. More strikingly, both applications might have used completely different platform resources (being assigned to different workers) and would have run concurrently at the same speed as if in an exclusive mode on the platform. Sharing resources to execute several applications concurrently has two key advantages: (i) from the clients' point of view, the average flow time (the delay between the arrival of an application and the completion of its last task) is

expected to be much smaller; (ii) from the resource usage perspective, different applications will have different characteristics, and are likely to be assigned different resources by the scheduler. Overall, the global utilization of the platform will increase. The traditional measure to quantify the benefits of concurrent scheduling on shared resources is the maximum stretch (Section 2.7). The objective is then to minimize the maximum stretch of any application, thereby enforcing a fair trade-off between all applications.

In a second part, we go back to a single fully parallel application, but we introduce a much more complex (and more realistic) framework than scheduling on a platform where each processor has a unique speed. We suppose here that all processors have several speeds (or modes) of computation. In this context, we would like to maximize the throughput while minimizing the energy consumed. Unfortunately, throughput can be maximized by using more energy to speed up processors, while energy can be minimized by reducing the speed of processors and so the total throughput. So the goals of low power consumption and high schedule quality are contradictory. Thus, power-aware scheduling is a bicriteria optimization problem and our goal becomes finding non-dominated schedules, i.e., those such that no schedule can both have higher throughput and use less energy. A common approach to bicriteria problems is to fix one of the parameters. This gives two interesting special cases. If we fix energy, we get the laptop problem, which asks "What is the best schedule achievable using a particular energy budget, before the 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?". We study these two problems first at processor's level, then at the platform's level. The first theoretical results obtained was an optimal greedy algorithms for both problems when the power consumption model is simple. If we introduce a more realistic model, then the problems become NP-hard.

The organization of this chapter is the following. Section 5.2 describes the platform and application models. Section 5.3 is an introduction to steady-state scheduling, and explain how to schedule a single bag-of-tasks application, and retrieve a asymptotically optimal solution.

Section 5.4 is devoted to the scheduling of multiple bag-of-tasks applications. We present the optimal solution for the offline case (Section 5.4.2) and heuristics for the online case (Section 5.4.3). In Section 5.4.4 we report an extensive set of simulations and MPI experiments, and we compare the optimal solution against several classical heuristics from the literature. Section 5.4.5 is devoted to an overview of related work.

Then Section 5.5 is devoted to the power consumption minimization problem. We present different power consumption models in Section 5.5.1. We study our problem at the processor level in Section 5.5.2 before extending it to the system level in Section 5.5.3. We introduce realistic power consumption models in Section 5.5.4. At last, Section 5.5.5 is devoted to an overview of related work.

Finally, we state some concluding remarks in Section 5.6.

5.2 Framework

In this chapter, we target a heterogeneous master-worker platform, under the bounded multi-port model. The link between P_{master} and P_u has a bandwidth b_u, and the bound on the amount of data that the master can send per time-unit is denoted by BW. We assume a linear cost model, hence it takes X/b_u time-units to send (resp. receive) a message of size X to (resp. from) P_u. The computational speed of worker P_u is s_u, meaning that it takes X/s_u time-units to execute

X floating point operations. We suppose we are under the **BMP-FC-SS** computation model (cf. Section 2.8), which provides an upper bound on the achievable performance for any other model, and is not too far from reality when considering small-size tasks.

Application model

We consider bag-of-tasks applications A_k, $1 \leq k \leq n$. The master P_{master} holds the input data of each application A_k upon its release time $r^{(k)}$. Application A_k is composed of a set of $\Pi^{(k)}$ independent, same-size tasks. In order to completely execute an application, all its constitutive tasks must be computed (in any order).

We let $w^{(k)}$ be the amount of computations (expressed in flops) required to process a task of A_k. The speed of a worker P_u may well be different for each application, depending upon the characteristics of the processor and upon the type of computations needed by each application. To take this into account, we refine the platform model and denote by $s_u^{(k)}$ the speed of worker P_u when processing application A_k. In other words, we suppose we are dealing with unrelated machines (cf. Section 2.3.1). The time required to process one task of A_k on processor P_u is thus $w^{(k)}/s_u^{(k)}$. Each task of A_k has a size $\delta^{(k)}$ (expressed in bytes), which means that it takes a time $\delta^{(k)}/b_u$ to send a task of A_k to processor P_u (when there are no other ongoing transfers). For simplicity we do not consider any return message: either we assume that the results of the tasks are stored on the workers, or we merge the return message of the current task with the input message of the next one (and update the communication volume accordingly).

5.3 Scheduling a single bag-of-tasks application

Assume for a while that a unique bag-of-tasks application A_k is executed on the platform. One can remark that this problem is similar to scheduling independent tasks without release dates, as studied in Section 3.3.3, and proved polynomial. Here we use the steady-state approach to solve this problem.

If $\Pi^{(k)}$, the number of independent tasks composing the application, is large (otherwise, why would we deploy A_k on a parallel platform?), we can relax the problem of minimizing the total execution time. Instead, we aim at maximizing the throughput, i.e., the average (fractional) number of tasks executed per time-unit. We design a cyclic schedule, that reproduces the same schedule every period, except possibly for the very first periods (initialization) and the last ones (clean-up). It is shown in [14, 9] how to derive an optimal schedule for throughput maximization. The idea is to characterize the optimal throughput as the solution of a linear program over rational numbers, which is a problem with polynomial time complexity.

Throughout this chapter, we denote by $\rho_u^{(k)}$ the throughput of worker P_u for application A_k, i.e., the average number of tasks of A_k that P_u executes per time-unit. We write the following linear program (see Equation (5.1)), which enables us to compute an asymptotically optimal schedule. The maximization of the throughput is bounded by three types of constraints:

- The first set of constraints states that the processing capacity of P_u is not exceeded.
- The second set of constraints states that the bandwidth of the link from P_{master} to P_u is not exceeded.
- The last constraint states that the total outgoing capacity of the master is not exceeded.

$$\begin{cases} \text{MAXIMIZE } \rho^{(k)} = \sum_{u=1}^{p} \rho_u^{(k)} \text{ SUBJECT TO} \\ \forall 1 \leq u \leq p, \quad \rho_u^{(k)} \frac{w^{(k)}}{s_u^{(k)}} \leq 1 \\ \forall 1 \leq u \leq p, \quad \rho_u^{(k)} \frac{\delta^{(k)}}{b_u} \leq 1 \\ \sum_{u=1}^{p} \rho_u^{(k)} \frac{\delta^{(k)}}{\text{BW}} \leq 1 \end{cases} \tag{5.1}$$

The formulation in terms of a linear program is simple when considering a single application. In this case, a closed-form expression can even be derived. First, the first two sets of constraints can be transformed into:

$$\forall 1 \leq u \leq p \quad \rho_u^{(k)} \leq \min \left\{ \frac{s_u^{(k)}}{w^{(k)}}, \frac{b_u}{\delta^{(k)}} \right\}.$$

Then, the last constraint can be rewritten:

$$\sum_{u=1}^{p} \rho_u^{(k)} \leq \frac{\text{BW}}{\delta^{(k)}}.$$

So that the total optimal throughput is

$$\rho^{(k)} = \min \left\{ \frac{\text{BW}}{\delta^{(k)}}, \sum_{u=1}^{p} \min \left\{ \frac{s_u^{(k)}}{w^{(k)}}, \frac{b_u}{\delta^{(k)}} \right\} \right\}.$$

It can be shown [14, 9] that any feasible schedule under one of the multiport models has to enforce the previous constraints. Hence the optimal value $\rho^{(k)}$ is an upper bound of the achievable throughput. Moreover, we can construct an actual schedule, based on an optimal solution of the linear program, and which approaches the optimal throughput. The reconstruction is particularly easy. For example the following procedure builds an asymptotic optimal schedule for the BMP-AC model (bounded multiport communications with atomic computations). As this is the most constrained multiport model, this schedule is feasible in any multiport model:

- While there are tasks to be processed on the master, send tasks to processor P_u with rate $\rho_u^{(k)}$.
- As soon as processor P_u receives a task it processes it at the rate $\rho_u^{(k)}$.

Due to the constraints of the linear program, this schedule is always feasible and it is asymptotically optimal, not only among periodic schedules, but more generally among any possible schedules. More precisely, its execution time differs from the minimum execution time by a constant factor, independent of the total number of tasks $\Pi^{(k)}$ to process [9]. This allows us to accurately approximate the total execution time by:

$$MS^{(k)} = \frac{\Pi^{(k)}}{\rho^{(k)}}.$$

We will often use $MS^{(k)}$ as a comparison basis to approximate the real makespan of an application when it is alone on the computing platform. If $MS^{(k)}_{\text{opt}}$ is the optimal makespan for this single application, then we have

$$MS^{(k)}_{\text{opt}} - M_k \leq MS^{(k)} \leq MS^{(k)}_{\text{opt}}$$

where M_k is a fixed constant [9], independent of $\Pi^{(k)}$.

However, as our fluid model requires a total control of the rate of computation and communication, it is quite hard to implement. In practice, the master uses the 1-D load balancing algorithm (please refer to Appendix C for more details about 1-D load balancing): when each participating worker P_u has already received n_u tasks, the next worker to receive a task is chosen as the one minimizing $\frac{n_u + 1}{\rho_u^{(k)}}$

5.4 Scheduling multiple bag-of-tasks applications

In this first part, we suppose that several applications are executed concurrently. Because they compete for resources, their throughputs will be lower. Equivalently, their executions rate will be slowed down. Informally, the stretch of an application is its slowdown factor.

5.4.1 Stretch

We denote by $\rho_u^{*(k)}$ the value of throughput of the application A_k when executed alone on the platform, and $MS^{*(k)}$ its makespan.

In our multiple bag-of-tasks applications context, the completion time of A_k is $\mathcal{C}^{(k)} = r^{(k)} + MS^{(k)}$, where $r^{(k)}$ is the release date of A_k, and $MS^{(k)}$ is the time to execute all $\Pi^{(k)}$ tasks of A_k. Because there might be other applications running concurrently to A_k during part or whole of its execution, we expect that $MS^{(k)} \geq MS^{*(k)}$. We define the average throughput $\rho^{(k)}$ achieved by A_k during its (concurrent) execution using the same equation as before:

$$MS^{(k)} = \frac{\Pi^{(k)}}{\rho^{(k)}}.$$

In order to process all applications fairly, we would like to ensure that their actual (concurrent) execution is as close as possible to their execution in dedicated mode. The stretch of application A_k is its slowdown factor

$$\mathcal{S}_k = \frac{MS^{(k)}}{MS^{*(k)}} = \frac{\rho^{*(k)}}{\rho^{(k)}}.$$

Our objective function is defined as the *max-stretch* $\mathcal{S}$, which is the maximum of the stretches of all applications:

$$\mathcal{S} = \max_{1 \leq k \leq n} \mathcal{S}_k.$$

Minimizing the *max-stretch* $\mathcal{S}$ ensures that the slowdown factor is kept as low as possible for each application, and that none of them is unduly favored by the scheduler.

5.4.2 Offline setting for the fluid model

In this section we present an asymptotically optimal polynomial algorithm to schedule several bag-of-tasks applications, while minimizing the maximum stretch, in the offline case. We assume that the release dates and characteristics of the n applications, A_k, $1 \leq k \leq n$, are known in advance.

Set of possible schedules

Given a candidate value for the max-stretch, we have a procedure to determine whether there exists a solution that can achieve this value. The optimal value will then be found using a binary search on possible values.

Consider a candidate value $\mathcal{S}^l$ for the max-stretch. If this objective is feasible, all applications will have a max-stretch no greater than $\mathcal{S}^l$, hence:

$$\begin{aligned} & \forall\ 1 \leq k \leq n, \quad \frac{MS^{(k)}}{MS^{*(k)}} \leq \mathcal{S}^l \\ \iff \quad & \forall\ 1 \leq k \leq n, \quad \mathcal{C}^{(k)} = r^{(k)} + MS^{(k)} \leq r^{(k)} + \mathcal{S}^l \times MS^{*(k)} \end{aligned}$$

Thus, given a candidate value $\mathcal{S}^l$, we have for each application A_k, $1 \leq k \leq n$ a deadline:

$$d^{(k)} = r^{(k)} + \mathcal{S}^l \times MS^{*(k)}. \tag{5.2}$$

This means that the application must complete no later than this deadline in order to ensure the expected max-stretch. If this is not possible, no solution is found, and a larger max-stretch should be tried by the binary search.

Once a candidate stretch value $\mathcal{S}$ has been chosen, we divide the total execution time into time-intervals whose bounds are epochal times, that is, applications' release dates or deadlines. Epochal times are denoted $t_j \in \{r^{(1)}, ..., r^{(n)}\} \cup \{d^{(1)}, \ldots, d^{(n)}\}$, such that $t_j \leq t_{j+1}$, $1 \leq j \leq 2n-1$. Our algorithm consists in running each application A_k during its whole execution window $[r^{(k)}, d^{(k)}]$, but with a different throughput on each time-interval $[t_j, t_{j+1}]$ such that $r^{(k)} \leq t_j$ and $t_{j+1} \leq d^{(k)}$. Some release dates and deadlines may be equal, leading to empty time-intervals, for example if there exists j such that $t_j = t_{j+1}$. We do not try to remove these empty time-intervals so as to keep simple indices.

Note that contrarily to the steady-state operation with only one application, in the different time-intervals, the communication throughput may differ from the computation throughput: when the communication rate is larger than the computation rate, extra tasks are stored in buffers. On the contrary, when the computation rate is larger, tasks are extracted from the buffers and processed. We introduce new notations to take both rates, as well as buffer sizes, into account:

- $\rho^{(k)}_{M \to u}(t_j, t_{j+1})$ denotes the communication throughput from the master to the worker P_u during time-interval $[t_j, t_{j+1}]$ for application A_k, i.e., the average number of tasks of A_k sent to P_u per time-units.
- $\rho^{(k)}_u(t_j, t_{j+1})$ denotes the computation throughput of worker P_u during time-interval $[t_j, t_{j+1}]$ for application A_k, i.e., the average number of tasks of A_k computed by P_u per time-units.
- $B^{(k)}_u(t_j)$ denotes the (fractional) number of tasks of application A_k stored in a buffer on P_u at time t_j.

We write the linear constraints that must be satisfied by the previous variables. Our aim is to find a schedule with minimum stretch satisfying those constraints.

All tasks sent by the master. The first set of constraints ensures that all the tasks of a given application A_k are actually sent by the master:

$$\forall\ 1 \leq k \leq n, \sum_{\substack{1 \leq j \leq 2n-1 \\ t_j \geq r^{(k)} \\ t_{j+1} \leq d^{(k)}}} \sum_{u=1}^{p} \rho_{M \to u}^{(k)}(t_j, t_{j+1}) \times (t_{j+1} - t_j) = \Pi^{(k)}. \tag{5.3}$$

Non-negative buffers. Each buffer should always have a non-negative size:

$$\forall\ 1 \leq k \leq n, \forall 1 \leq u \leq p, \forall 1 \leq j \leq 2n, \quad B_u^{(k)}(t_j) \geq 0. \tag{5.4}$$

Buffer initialization. At the beginning of the computation of application A_k, all corresponding buffers are empty:

$$\forall\ 1 \leq k \leq n, \forall 1 \leq u \leq p, \quad B_u^{(k)}(r^{(k)}) = 0. \tag{5.5}$$

Emptying Buffer. After the deadline of application A_k, no tasks of this application should remain on any node:

$$\forall\ 1 \leq k \leq n, \forall 1 \leq u \leq p, \quad B_u^{(k)}(d^{(k)}) = 0. \tag{5.6}$$

Task conservation. During time-interval $[t_j, t_{j+1}]$, some tasks of application A_k are received and some are consumed (computed), which impacts the size of the buffer:

$$\begin{aligned} &\forall\ 1 \leq k \leq n, \forall 1 \leq j \leq 2n-1, \forall 1 \leq u \leq p, \\ &\qquad B_u^{(k)}(t_{j+1}) = B_u^{(k)}(t_j) + \left(\rho_{M \to u}^{(k)}(t_j, t_{j+1}) - \rho_u^{(k)}(t_j, t_{j+1})\right) \times \left(t_{j+1} - t_j\right). \end{aligned} \tag{5.7}$$

Bounded computing capacity. The computing capacity of a node should not be exceeded on any time-interval:

$$\forall 1 \leq j \leq 2n-1, \forall 1 \leq u \leq p, \ \sum_{k=1}^{n} \rho_u^{(k)}(t_j, t_{j+1}) \frac{w^{(k)}}{s_u^{(k)}} \ \leq \ 1. \tag{5.8}$$

Bounded link capacity. The bandwidth of each link should not be exceeded:

$$\forall 1 \leq j \leq 2n-1, \forall 1 \leq u \leq p, \ \sum_{k=1}^{n} \rho_{M \to u}^{(k)}(t_j, t_{j+1}) \frac{\delta^{(k)}}{b_u} \leq \ 1. \tag{5.9}$$

Limited sending capacity of master. The total outgoing bandwidth of the master should not be exceeded:

$$\forall 1 \leq j \leq 2n-1, \quad \sum_{u=1}^{p} \sum_{k=1}^{n} \rho_{M \to u}^{(k)}(t_j, t_{j+1}) \frac{\delta^{(k)}}{\mathrm{BW}} \leq 1. \tag{5.10}$$

Non-negative throughputs.

$$\forall 1 \leq u \leq p, \forall 1 \leq k \leq n, \forall 1 \leq j \leq 2n-1, \quad \rho^{(k)}_{M\to u}(t_j, t_{j+1}) \geq 0 \text{ and } \rho^{(k)}_{u}(t_j, t_{j+1}) \geq 0. \tag{5.11}$$

We obtain a convex polyhedron (K) defined by the previous constraints. The problem turns now into checking whether the polyhedron is empty and, if not, into finding a point in the polyhedron.

$$\begin{cases} \rho^{(k)}_{M\to u}(t_j, t_{j+1}), \rho^{(k)}_{u}(t_j, t_{j+1}), \ \forall k, u, j \text{ such that } 1 \leq k \leq n, 1 \leq u \leq p, 1 \leq j \leq 2n-1 \\ \text{under the constraints (5.3), (5.7), (5.5), (5.6), (5.4), (5.8), (5.9), (5.10) and (5.11)} \end{cases} \tag{K}$$

Number of tasks processed

At first sight, it may seem surprising that in this set of linear constraints, we do not have an equation establishing that all tasks of a given application are eventually processed. Indeed, such a constraint can be derived from the constraints related to the number of tasks sent from the master and the size of buffers. Consider the constraints on task conservation (Equation (5.7)) on a given processor P_u, and for a given application A_k; these equations can be written:

$$\forall 1 \leq j \leq 2n-1, \quad B^{(k)}_u(t_{j+1}) - B^{(k)}_u(t_j) = \left(\rho^{(k)}_{M\to u}(t_j, t_{j+1}) - \rho^{(k)}_u(t_j, t_{j+1})\right) \times \left(t_{j+1} - t_j\right).$$

If we sum all these constraints for all time-interval bounds between $t_{\text{start}} = r^{(k)}$ and $t_{\text{stop}} = d^{(k)}$, we obtain:

$$B^{(k)}_u(t_{\text{stop}}) - B^{(k)}_u(t_{\text{start}}) = \sum_{\substack{[t_j,\ t_{j+1}] \\ t_j \geq r^{(k)} \\ t_{j+1} \leq d^{(k)}}} \left(\rho^{(k)}_{M\to u}(t_j, t_{j+1}) - \rho^{(k)}_u(t_j, t_{j+1})\right) \times \left(t_{j+1} - t_j\right).$$

Thanks to constraints (5.5) and (5.6), we know that $B^{(k)}_u(t_{\text{start}}) = 0$ and $B^{(k)}_u(t_{\text{stop}}) = 0$. So the overall number of tasks sent to a processor P_u is equal to the total number of tasks computed:

$$\sum_{\substack{[t_j,\ t_{j+1}] \\ t_j \geq r^{(k)} \\ t_{j+1} \leq d^{(k)}}} \rho^{(k)}_{M\to u}(t_j, t_{j+1}) \times \left(t_{j+1} - t_j\right) = \sum_{\substack{[t_j,\ t_{j+1}] \\ t_j \geq r^{(k)} \\ t_{j+1} \leq d^{(k)}}} \rho^{(k)}_{u}(t_j, t_{j+1}) \times \left(t_{j+1} - t_j\right).$$

This is true for all processors, and constraints (5.3) tells us that the total number of tasks sent for application A_k is $\Pi^{(k)}$, so:

$$\sum_{u=1}^{p} \sum_{\substack{[t_j,\ t_{j+1}] \\ t_j \geq r^{(k)} \\ t_{j+1} \leq d^{(k)}}} \rho^{(k)}_{u}(t_j, t_{j+1}) \times \left(t_{j+1} - t_j\right) = \Pi^{(k)}$$

Therefore in any solution in Polyhedron (K), all tasks of each application are processed.

Bounding the buffer size

The size of the buffers could also be bounded by adding constraints:

$$\forall 1 \leq u \leq p, \forall 1 \leq j \leq 2n, \quad \sum_{k=1}^{n} B_u^{(k)}(t_j)\delta^{(k)} \leq M_u$$

where M_u is the size of the memory available on node P_u. We bound the needed memory only at time-interval bounds, but the above argument can be used to prove that the buffer size on P_u never exceeds M_u. We choose not to include this constraint in our basic set of constraints, as this buffer size limitation only applies to the fluid model. Indeed, it has been proved that limiting the buffer size for independent tasks scheduling leads to NP-complete problems [20].

Equivalence between non-emptiness of Polyhedron (K) and achievable stretch

Finding a point in Polyhedron (K) allows to determine whether the candidate value for the stretch is feasible. Depending on whether Polyhedron (K) is empty, the binary search will be continued with a larger or smaller stretch value:

- If the polyhedron is not empty, then there exists a schedule achieving stretch $\mathcal{S}$. $\mathcal{S}$ becomes the upper bound of the binary search interval and the search proceeds.
- On the contrary, if the polyhedron is empty, then it is not possible to achieve $\mathcal{S}$. $\mathcal{S}$ becomes the lower bound of the binary search.

This binary search and its proof are described below. For now, we concentrate on proving that the polyhedron is not empty if and only if the stretch $\mathcal{S}$ is achievable.

Note that the previous study assumes a fluid framework, with flexible computing and communicating rates. This is particularly convenient for the totally fluid model (BMP-FC-SS) and we prove below that the algorithm computes the optimal stretch under this model. The strength of our method is that this study is also valid for the other models. The results are slightly different, leading to asymptotic optimality results and the proofs detailed below are slightly more involved. However, this technique allows to approach optimality. One can find in Appendix C how to construct a schedule achieving the corresponding stretch based on rates satisfying the totally fluid model for all other platform models.

Theorem 5.1. *Under the totally fluid model, Polyhedron (K) is not empty if and only if there exists a schedule with stretch $\mathcal{S}$.*

Proof. $\boxed{\Rightarrow}$ Assume that the polyhedron is not empty, and consider a point in (K), given by the values of the $\rho_{M\to u}^{(k)}(t_j, t_{j+1})$ and $\rho_u^{(k)}(t_j, t_{j+1})$. We construct a schedule which obeys exactly these values. During time-interval $[t_j, t_{j+1}]$, the master sends tasks of application A_k to processor P_u with rate $\rho_{M\to u}^{(k)}(t_j, t_{j+1})$, and this processor computes these tasks at a rate $\rho_u^{(k)}(t_j, t_{j+1})$.

To prove that this schedule is valid under the fluid model, and that it has the expected stretch, we define $\rho_{M\to u}^{(k)}(t)$ as the instantaneous communication rate, and $\rho_u^{(k)}(t)$ as the instantaneous computation rate. Then the (fractional) number of tasks of A_k sent to P_u in interval $[0, T]$ is

$$\int_0^T \rho_{M\to u}^{(k)}(t)dt$$

With the same argument as in the previous remark, applied on interval $[0, T]$, we have

$$B_u^{(k)}(T) = \int_0^T \rho_{M\to u}^{(k)}(t)dt - \int_0^T \rho_u^{(k)}(t)dt.$$

Since the buffer size is positive for all t_j and evolves linearly in each interval $[t_j, t_{j+1}]$, it is not possible that a buffer has a negative size, so

$$\int_0^T \rho_u^{(k)}(t)dt \leq \int_0^T \rho_{M\to u}^{(k)}(t)dt$$

Hence data is always received before being processed.

With the constraints of Polyhedron (K), it is easy to check that no processor or no link is over-utilized and the outgoing capacity of the master is never exceeded. All the deadlines computed for stretch $\mathcal{S}$ are satisfied by construction, so this schedule achieves stretch $\mathcal{S}$.

$\boxed{\Leftarrow}$ Now we prove that if there exists a schedule S_1 with stretch $\mathcal{S}$, Polyhedron (K) is not empty. We consider such a schedule, and we call $\rho_{M\to u}^{(k)}(t)$ (and $\rho_u^{(k)}(t)$) the communication (and computation) rate in this schedule for tasks of application A_k on processor P_u at time t. We compute as follows the average values for communication and computation rates during time interval $[t_j, t_{j+1}]$:

$$\rho_{M\to u}^{(k)}(t_j, t_{j+1}) = \frac{\displaystyle\int_{t_j}^{t_{j+1}} \rho_{M\to u}^{(k)}(t)dt}{t_{j+1} - t_j} \text{ and } \rho_u^{(k)}(t_j, t_{j+1}) = \frac{\displaystyle\int_{t_j}^{t_{j+1}} \rho_u^{(k)}(t)dt}{t_{j+1} - t_j}.$$

Of course, if $t_{j+1} = t_j$, we just set $\rho_{M\to u}^{(k)}(t_j, t_{j+1}) = \rho_u^{(k)}(t_j, t_{j+1}) = 0$. In this schedule, all tasks of application A_k are sent by the master, so

$$\int_{r^{(k)}}^{d^{(k)}} \rho_{M\to u}^{(k)}(t)dt = \Pi^{(k)}.$$

With the previous definitions, Equation (5.3) is satisfied. Along the same line, we can prove that the task conservation constraints (Equation (5.7)) are satisfied. Constraints on buffers (Equations 5.4, 5.5, and 5.6) are necessarily satisfied by the size of the buffer in schedule S_1 since it is feasible. Similarly, we can check that the constraints on capacities are verified. ■

In practice, to know if the polyhedron is empty or to obtain a point in (K), we can use classical tools for linear programs, just by adding a fictitious linear objective function to our set of constraints. Some solvers allow the user to limit the number of refinement steps once a point is found in the polyhedron; this could be helpful to reduce the running time of the scheduler.

Binary search

To find the optimal stretch, we perform a binary search. We first present a simple approximated search using the emptiness of Polyhedron (K) to determine whether it is possible to achieve the current stretch. Then we present an optimal but more involved search.

The lower bound on the achievable stretch is 1. The initial upper bound for this binary search is also quite naive. For the sake of simplicity, we consider that all applications are released at time 0 and terminate simultaneously. This is clearly a worst case scenario. Recall that the optimal throughput for a single application on the whole platform can be computed as:

$$\rho^{*(k)} = \min\left\{\frac{\mathrm{BW}}{\delta^{(k)}}, \sum_{u=1}^{p} \min\left\{\frac{s_u^{(k)}}{w^{(k)}}, \frac{b_u}{\delta^{(k)}}\right\}\right\}$$

Then the execution time for application A_k is simply underestimated by $\Pi^{(k)}/\rho^{*(k)}$. We consider that all applications terminate at time $\sum_k \Pi^{(k)}/\rho^{*(k)}$, so that the worst stretch is

$$\mathcal{S}_{\max} = \max_k \frac{\Pi^{(k)}/\rho^{*(k)}}{\sum_j \Pi^{(j)}/\rho^{*(j)}}.$$

Determining the termination criterion of the binary search, that is the minimum gap ϵ between two possible stretches, is quite involved, and not very useful in practice. We focus here on the case where this precision ϵ is given by the user. We will later see a low-complexity technique (a binary search among stretch-intervals) to compute the optimal maximum stretch.

Suppose that we are given $\epsilon > 0$. The binary search is conducted using Algorithm 9. This algorithm allows us to approach the optimal stretch, as stated by the following theorem.

Algorithm 9: Binary search

```
begin
    S_inf ← 1
    S_sup ← S_max
    while S_sup − S_inf > ε do
        S ← (S_sup + S_inf)/2
        if Polyhedron (K) is empty then
            S_inf ← S
        else
            S_sup ← S
    return S_sup
end
```

Theorem 5.2. *For any $\epsilon > 0$, Algorithm 9 computes a stretch $\mathcal{S}$ such that there exists a schedule achieving $\mathcal{S}$ and $\mathcal{S} \leq \mathcal{S}_{\mathrm{opt}} + \epsilon$, where $\mathcal{S}_{\mathrm{opt}}$ is the optimal stretch. The complexity of Algorithm 9 is $O\left(\log \frac{\mathcal{S}_{\max}}{\epsilon}\right)$.*

Proof. We prove that at each step, the optimal stretch is contained in the interval $[\mathcal{S}_{\mathrm{inf}}, \mathcal{S}_{\mathrm{sup}}]$ and $\mathcal{S}_{\mathrm{sup}}$ is achievable. This is obvious at the beginning. At each step, we consider the set of constraints for a stretch $\mathcal{S}$ in the interval. If the corresponding polyhedron is empty, Theorem 5.1 tells us that stretch $\mathcal{S}$ is not achievable, so the optimal stretch is greater than $\mathcal{S}$. If the polyhedron is not empty, there exists a schedule achieving this stretch, thus the optimal stretch is smaller than $\mathcal{S}$.

The size of the work interval is divided by 2 at each step, and we stop when this size is smaller than ϵ. Thus the number of steps is $O\left(\log \frac{\mathcal{S}_{max}}{\epsilon}\right)$. At the end, $\mathcal{S}_{\mathrm{opt}} \in [\mathcal{S}_{\mathrm{inf}}, \mathcal{S}_{\mathrm{sup}}]$ with $\mathcal{S}_{\mathrm{sup}} - \mathcal{S}_{\mathrm{inf}} \leq \epsilon$, so that $\mathcal{S}_{\mathrm{sup}} \leq \mathcal{S}_{\mathrm{opt}} + \epsilon$, and $\mathcal{S}_{\mathrm{sup}}$ is achievable. ■

Binary search with stretch-intervals

In this section, we present another method which computes the optimal stretch in the offline case. This method is based on a linear program built from the constraints of the convex polyhedron (K)

with the minimization of the stretch as objective. To do this, we need that other parameters (especially the deadlines) are functions of the stretch. We recall that the deadlines of the applications are computed from their release date and the targeted stretch $\mathcal{S}$:

$$d^{(k)} = r^{(k)} + \mathcal{S} \times MS^{*(k)}.$$

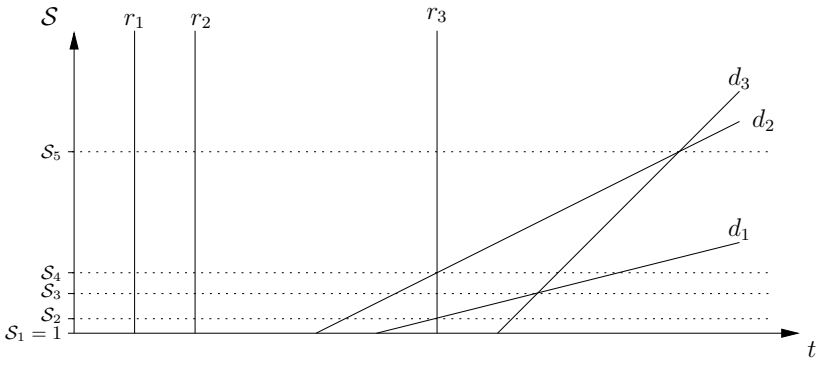


Figure 5.1: Relation between stretch and deadlines

Figure 5.1 represents the evolution of the deadlines $d^{(k)}$ over the targeted stretch $\mathcal{S}$: each deadline is an affine function in $\mathcal{S}$. For the sake of readability, the time is represented on the x axis, and the stretch on the y axis. Special values of stretches $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ are represented in the figure. These *critical values* of the stretch are points where the ordering of the applications' release dates and deadlines is modified:

- When $\mathcal{S}$ is such a *critical value*, some release dates and deadlines have the same value;
- When $\mathcal{S}$ varies between two such critical values, i.e., when $\mathcal{S}_a < \mathcal{S} < \mathcal{S}_{a+1}$, then the ordering of the release dates and the deadlines is preserved.

To simplify our notations, we add two artificial *critical values* corresponding to the natural bound of the stretch: $\mathcal{S}_1 = 1$ and $\mathcal{S}_m = \infty$.

Our goal is to find the optimal stretch by slicing the stretch space into a number of intervals. Within each interval defined by the *critical values*, the deadlines are affine functions of the stretch. We first show how to find the best stretch within a given interval using a single linear program, and then how to explore the set of intervals with a binary search, so as to find the one containing the optimal stretch.

Within a stretch-interval

In the following, we work on one stretch-interval, called $[\mathcal{S}_a, \mathcal{S}_b]$. For all values of $\mathcal{S}$ in this interval, the release dates $r^{(k)}$ and deadlines $d^{(k)}$ are in a given order, independent of the value of $\mathcal{S}$. As previously, we note $\{t_j\}_{j=1\ldots 2n} = \{r^{(k)}, d^{(k)}\}_k$, with $t_j \leq t_{j+1}$. As the values of the t_j may change when $\mathcal{S}$ varies, we write $t_j = \alpha_j \mathcal{S} + \beta_j$. This notation is general enough for all $r^{(k)}$'s and $d^{(k)}$'s:

- If $t_j = r^{(k)}$, then $\alpha_j = 0$ and $\beta_j = r^{(k)}$;
- If $t_j = d^{(k)}$, then $\alpha_j = MS^{*(k)}$ and $\beta_j = r^{(k)}$.

Note that like previously, some t_j might be equal, and especially when the stretch reaches a bound of the stretch-interval ($\mathcal{S} = \mathcal{S}_a$ or $\mathcal{S} = \mathcal{S}_b$), that is a critical value. For the sake of simplicity, we do not try to discard the empty time-intervals, to avoid renumbering the epochal times.

When we rewrite the constraints defining the convex polyhedron (K) with these new notations, we obtain quadratic constraints instead of linear constraints. To avoid this, we introduce new notations. Instead of considering the instantaneous communication and computation rates, we use the total amount of tasks sent or computed during a given time-interval. Formally we define $A^{(k)}_{M\to u}(t_j, t_{j+1})$ to be the fractional number of tasks of application A_k sent by the master to processor P_u during the time-interval $[t_j, t_{j+1}]$. Similarly, we denote by $A^{(k)}_u(t_j, t_{j+1})$ the fractional number of tasks of application A_k computed by processor P_u during the time-interval $[t_j, t_{j+1}]$. Of course, these quantities are linked to our previous variables. Indeed, we have:

$$\begin{aligned} A^{(k)}_{M\to u}(t_j, t_{j+1}) &= \rho^{(k)}_{M\to u}(t_j, t_{j+1}) \times (t_{j+1} - t_j) \\ A^{(k)}_u(t_j, t_{j+1}) &= \rho^{(k)}_u(t_j, t_{j+1}) \times (t_{j+1} - t_j) \end{aligned}$$

with $t_{j+1} - t_j = (\alpha_{j+1} - \alpha_j)\mathcal{S} + (\beta_{j+1} - \beta_j)$.

All constraints can be rewritten with these new notations:

Total number of tasks. We make sure that all tasks of application A_k are sent by the master:

$$\forall\ 1 \le k \le n, \qquad \sum_{\substack{1\le j\le 2n-1 \\ t_j \ge r^{(k)} \\ t_{j+1} \le d^{(k)}}} \sum_{u=1}^{p} A^{(k)}_{M\to u}(t_j, t_{j+1}) = \Pi^{(k)}. \tag{5.12}$$

Non-negative buffer. Each buffer should always have a non-negative size:

$$\forall\ 1 \le k \le n, \forall 1 \le u \le p, \forall 1 \le j \le 2n, \quad B^{(k)}_u(t_j) \ge 0. \tag{5.13}$$

Buffer initialization. At the beginning of the computation of application A_k, all corresponding buffers are empty:

$$\forall\ 1 \le k \le n, \forall 1 \le u \le p, \quad \text{for } t_j = r^{(k)}, \quad B^{(k)}_u(t_j) = 0. \tag{5.14}$$

Emptying Buffer. After the deadline of application A_k, no tasks of this application should remain on any node:

$$\forall\ 1 \le k \le n, \forall 1 \le u \le p, \quad \text{for } t_j = d^{(k)}, \quad B^{(k)}_u(t_j) = 0. \tag{5.15}$$

Task conservation. During time-interval $[t_j, t_{j+1}]$, some tasks of application A_k are received and some are consumed (computed), which impacts the size of the buffer:

$$\begin{aligned} &\forall\ 1 \le k \le n, \forall 1 \le j \le 2n-1, \forall 1 \le u \le p, \\ &\qquad B^{(k)}_u(t_{j+1}) = B^{(k)}_u(t_j) + A^{(k)}_{M\to u}(t_j, t_{j+1}) - A^{(k)}_u(t_j, t_{j+1}). \end{aligned} \tag{5.16}$$

Bounded computing capacity. The computing capacity of a node should not be exceeded in any time-interval:

$$\forall 1 \leq j \leq 2n-1, \forall 1 \leq u \leq p, \quad \sum_{k=1}^{n} A_u^{(k)}(t_j, t_{j+1}) \frac{w^{(k)}}{s_u^{(k)}} \leq (\alpha_{j+1} - \alpha_j)\mathcal{S} + (\beta_{j+1} - \beta_j). \quad (5.17)$$

Bounded link capacity. The bandwidth of each link should not be exceeded:

$$\forall 1 \leq j \leq 2n-1, \forall 1 \leq u \leq p, \sum_{k=1}^{n} A_{M\to u}^{(k)}(t_j, t_{j+1}) \frac{\delta^{(k)}}{b_u} \leq (\alpha_{j+1} - \alpha_j)\mathcal{S} + (\beta_{j+1} - \beta_j). \quad (5.18)$$

Limited sending capacity of master. The total outgoing bandwidth of the master should not be exceeded:

$$\forall 1 \leq j \leq 2n-1, \sum_{u=1}^{p} \sum_{k=1}^{n} A_{M\to u}^{(k)}(t_j, t_{j+1}) \delta^{(k)} \leq \mathrm{BW} \times \big((\alpha_{j+1} - \alpha_j)\mathcal{S} + (\beta_{j+1} - \beta_j)\big). \quad (5.19)$$

We finally add a constraint to force the objective stretch to be in the targeted stretch-interval:

$$\mathcal{S}_a \leq \mathcal{S} \leq \mathcal{S}_b. \quad (5.20)$$

We thus rewrite as above all the constraints defining Polyhedron (K) and then we add the new constraint (5.20). This way, we obtain a linear program enabling us to check what is the minimal achievable stretch in the interval $[\mathcal{S}_a, \mathcal{S}_b]$, if any.

Even if the bounds of the sum on the time-intervals in Equation (5.12) seem to depend on $\mathcal{S}$, the set of intervals involved in the sum does not vary as the order of the t_j values is fixed for $\mathcal{S}_a \leq \mathcal{S} \leq \mathcal{S}_b$. With the objective of minimizing the stretch, we get the following linear program.

$$(\mathrm{LP}) \begin{cases} \text{Minimize } \mathcal{S}, \\ \text{under the constraints} \\ (5.12),\ (5.13),\ (5.14),\ (5.15),\ (5.16),\ (5.17),\ (5.18),\ (5.19),\ (5.20) \end{cases}$$

Solving this linear program allows to find the minimum possible stretch in the stretch-interval $[\mathcal{S}_a, \mathcal{S}_b]$. If the minimum stretch computed by the linear program is $\mathcal{S}_{\mathrm{opt}} > \mathcal{S}_a$, this means that there is not better possible stretch in $[\mathcal{S}_a, \mathcal{S}_b]$, and thus there is no better stretch for all possible values. On the contrary, if $\mathcal{S}_{\mathrm{opt}} = \mathcal{S}_a$, $\mathcal{S}_a$ may be the optimal stretch, or the optimal stretch may be smaller than $\mathcal{S}_a$. In this case, the binary search is continued with smaller stretch values. At last, if there is no solution to the linear program, then there exists no possible stretch smaller or equal to $\mathcal{S}_b$, and the binary search is continued with larger stretch values. This binary search and its proof are described below.

When $\mathcal{S}_a < \mathcal{S}_{\mathrm{opt}} \leq \mathcal{S}_b$, we can prove that $\mathcal{S}_{\mathrm{opt}}$ is the optimal stretch.

Theorem 5.3. *The linear program* (LP) *finds the optimal stretch provided that the optimal stretch is in* $]\mathcal{S}_a, \mathcal{S}_b]$.

Proof. The proof highly depends on Theorem 5.1. First, consider an optimal solution of the linear program (LP). We compute, for all u, k, j

$$\begin{aligned} \rho^{(k)}_{M\to u}(t_j, t_{j+1}) &= \frac{A^{(k)}_{M\to u}(t_j, t_{j+1})}{(\alpha_{j+1}-\alpha_j)\mathcal{S} + (\beta_{j+1}-\beta_j)} \\ \text{and} \quad \rho^{(k)}_{u}(t_j, t_{j+1}) &= \frac{A^{(k)}_{u}(t_j, t_{j+1})}{(\alpha_{j+1}-\alpha_j)\mathcal{S} + (\beta_{j+1}-\beta_j)}. \end{aligned}$$

These variables constitute a valid solution of the set of constraints of Theorem 5.1 for $\mathcal{S} = \mathcal{S}_{\text{opt}}$. Therefore there exists a schedule achieving stretch $\mathcal{S}_{\text{opt}}$.

Assume now that there exists a schedule with stretch $\mathcal{S}$ such that $\mathcal{S}_a < \mathcal{S} < \mathcal{S}_b$. Due to Theorem 5.1, there exist values for $\rho^{(k)}_{M\to u}(t_j, t_{j+1})$ and $\rho^{(k)}_{u}(t_j, t_{j+1})$ satisfying the corresponding set of constraints for $\mathcal{S}$. Then we compute

$$\begin{aligned} A^{(k)}_{M\to u}(t_j, t_{j+1}) &= \rho^{(k)}_{M\to u}(t_j, t_{j+1}) \times \big((\alpha_{j+1}-\alpha_j)\mathcal{S} + (\beta_{j+1}-\beta_j)\big) \\ \text{and} \quad A^{(k)}_{u}(t_j, t_{j+1}) &= \rho^{(k)}_{u}(t_j, t_{j+1}) \times \big((\alpha_{j+1}-\alpha_j)\mathcal{S} + (\beta_{j+1}-\beta_j)\big), \end{aligned}$$

$A^{(k)}_{M\to u}(t_j, t_{j+1})$ and $A^{(k)}_{u}(t_j, t_{j+1})$ constitute a solution of the linear program (LP) with objective value $\mathcal{S}$. As the objective value $\mathcal{S}_{\text{opt}}$ found by the linear program is minimal among all possible solutions, we have $\mathcal{S}_{\text{opt}} \leq \mathcal{S}$. ■

Binary search among stretch intervals

The linear program we just described is used as a building brick for our exact binary search between the critical values. As the number of critical values is at worst quadratic in the number of applications, the overall binary search runs in time polynomial in the size of the problem.

We assume that we have computed the critical values of the stretch intervals: $\mathcal{S}_1, \ldots, \mathcal{S}_m$. Algorithm 10 describes the binary search to reach the optimal stretch.

Theorem 5.4. *Algorithm 10 finds the optimal stretch value in a polynomial number of steps.*

Proof. This algorithm performs a binary search among the m stretch-intervals. Thus, the number of steps of this search is $O(\log m)$ and each step consists in solving a linear program with rational variables, which can be done in polynomial time.

We prove that the optimal stretch is always contained in the interval $[\mathcal{S}_L, \mathcal{S}_U]$. This is obviously true in the beginning. On a stretch-interval $[\mathcal{S}_M, \mathcal{S}_{M+1}]$, the minimum possible stretch $\mathcal{S}_{\text{opt}}$ is computed. If $\mathcal{S}_{\text{opt}} > \mathcal{S}_M$, thanks to Theorem 5.3, we know that $\mathcal{S}_{\text{opt}}$ is the optimal stretch. If there is no solution, no stretch values in the stretch-interval $[\mathcal{S}_M, \mathcal{S}_{M+1}]$ is feasible, so the optimal stretch is in $[\mathcal{S}_{M+1}, \mathcal{S}_U]$. If $\mathcal{S}_{\text{opt}} = \mathcal{S}_M$, then the optimal stretch is smaller than or equal to $\mathcal{S}_M$. Thus, the optimal stretch is still contained in $[\mathcal{S}_L, \mathcal{S}_U]$ after one iteration. If we exit *while* loop without having return the optimal stretch, then $U = L + 1$ and the optimal stretch is contained in the stretch-interval $[\mathcal{S}_L, \mathcal{S}_U]$. We compute this value with the linear program and return it. ■

Algorithm 10: Binary search among stretch-intervals

```
begin
    L ← 1 and U ← max
    while U − L > 1 do
        M ← ⌊(L + U)/2⌋
        Solve the linear program (LP) for interval [S_M, S_{M+1}]
        if there is a solution with objective value S_opt then
            if S_opt > S_M then
                return S_opt
            else
                U ← M
        else
            L ← M
    Solve the linear program (LP) for interval [S_L, S_U]
    return the objective value S_opt of the solution
end
```

5.4.3 Online setting

In this section we move to study the online setting, and we develop a polynomial time heuristic to schedule several bag-of-tasks applications arriving online, while minimizing the maximum stretch. Because we target an online framework, the scheduling policy needs to be modified upon the completion of an application, or upon the arrival of a new one. Resources will be re-assigned to the various applications in order to optimize the objective function. The scheduler is making best use of its partial knowledge of the whole process (we know beforehand neither the release dates, nor the number of tasks, nor the characteristics of the next application to arrive into the system). The idea is to make use of our study of the offline case. When a new application is released, we recompute the achievable max-stretch using the binary search described in the offline case. However, we cannot pretend to optimality any longer as we now have only limited information on the applications.

When a new application $A_{k_{\text{new}}}$ arrives at time $T_{\text{new}} = r^{(k_{\text{new}})}$, we consider the applications $A_0, \ldots, A_{k_{\text{new}}-1}$, released before T_{new}.
We call $\Pi_{\text{rem}}^{(k)}$ the (fractional) number of tasks of application A_k remaining at the master at time T_{new}. For the sake of simplicity, we do not consider the applications that are totally processed, and we thus have $\Pi_{\text{rem}}^{(k)} \neq 0$ for all applications. For the new application, we have $\Pi_{\text{rem}}^{(k_{\text{new}})} = \Pi^{(k_{\text{new}})}$. We also consider as parameters the state $B_u^{(k)}(t_{k_{\text{new}}})$ of the buffers at time T_{new}. We also have $B_u^{(k_{\text{new}})}(t_{k_{\text{new}}}) = 0$

As previously, we compute the optimal achievable max-stretch using Algorithm 9, the only slight modification being that we take past decisions into account. For a given objective $\mathcal{S}$, we have a convex polyhedron defined by the linear constraints, which is non empty if and only if stretch $\mathcal{S}$ is achievable. The constraints are slightly modified in order to fit the online context. First, we recompute the deadlines of the applications: $d^{(k)} = r^{(k)} + \mathcal{S} \times MS^{*(k)}$. Note that now, all release dates are smaller than T_{new}, and all deadlines are larger than T_{new}.

We sort the deadlines by increasing order, and denote by t_j the set of orderer deadlines: $\{t_j\} = \{d^{(k)}\} \cup \{T_{\text{new}}\}$ such that $t_j \leq t_{j+1}$. In other words, the constraints are the same as the

ones used for Polyhedron (K), except the constraint on the number of task processed, which is updated to account for the remaining number of tasks to be processed.

As described for the offline setting, a binary search allows to find the optimal max-stretch. Note that this "optimality" concerns only the time interval $[T_{\text{new}}, +\infty]$, assuming that no other application will be released after T_{new}. This assumption will not hold true in general, hence our schedule will be suboptimal (which is the price to pay without information about future released applications). The stretch achieved for the whole application set is bounded by the maximum of the stretches obtained by the binary search each time a new application is released.

5.4.4 MPI experiments and SimGrid simulations

We have conducted several experiments in order to compare different scheduling strategies, and to show the benefits of the heuristics presented in this section. We first present the heuristics compared. Then we detail the platforms and applications used for the experiments. Finally, we expose and comment the numerical results.

As our fluid model requires a total control of the rate of computation and communication, it is quite hard to implement in practice. During the experiments we used the **One-Port, Atomic Computation** model, which serializes sending from the master, and we assume that a worker has to completely receive a task before starting its computation, and that it cannot perform several computations simultaneously.

The code and the experimental results can be downloaded from:

http://graal.ens-lyon.fr/~jfpineau/Downloads/cbs3m/.

Heuristics

In this section, we present strategies that are able to schedule bag-of-tasks applications in an online setting. Most of these strategies are simple and wait for an application to terminate before scheduling another application. Although far from the optimal in a number of cases, such strategies are representative of existing Grid schedulers.

FIFO (First In First Out)– Applications are computed in the order of their release dates.

NPSPT (Non Preemptive Shortest Processing Time)– When an application terminates (or the first application is released), the application with the smallest processing time is scheduled (the processing time is approximated by MS^*, see Section 5.3).

SRPT (Shortest Remaining Processing Time)– At each release date or termination date, the application with the smallest remaining processing time is scheduled. The remaining processing time is the time needed to process the remaining tasks of the application (and is approximated as previously).

SWRPT (Shortest Weighted Remaining Processing Time)– This strategy is very similar to **SRPT**, but the remaining processing time of the released applications are weighted with MS^*, that is the application with the smallest product between the remaining processing time and the total processing time is scheduled first. In practice, it gives small applications a priority against large applications which are almost finished, which is better in order to minimize the stretch.

The importance and relevance of the above heuristics are outlined in the related work section (Section 5.4.5). Once an application is selected, several policies exist for scheduling its tasks onto the platform:

RR (Round-Robin)– All workers are selected in a cyclic way.

MCT (Minimum Completion Time)– Given the task, we select the worker that will finish this task the earliest, given the current load of the platform.

DD (Demand-Driven)– Workers are themselves asking for a task to compute as soon as they become idle.

The four application selection policies and the three resource selection rules lead to twelve different greedy algorithms. We also test a more sophisticated algorithm:

MWMA (Master Worker for Multiple Applications)– This algorithm computes on each time interval a steady-state strategy to schedule the available applications, as presented in [17] and [18]. All available applications are running at the same time, and each application is given a different fraction of the platform according to its weight. This weight can be derived from (i) the remaining number of tasks of the application (variant called **MWMA_NBT**), or (ii) the remaining time of computation of the application (variant called **MWMA_MS**). Both variants are studied in the experiments.

In addition to the previous scheduling strategies, we have implemented several heuristics based on our static algorithm, called **CBS3M** (for Clever Burst Steady-State Stretch Minimization) in the following. However, the use of MPI to perform communications during the experiments leads us to serialize the communications, and to abandon the multiport model in favor of the one-port model. Thus, the fluid solution of the **CBS3M** algorithm needs to be adapted to cope with the one-port model. Rather than literally implementing the transformations of Appendix C, which are best suited to compute theoretical bounds, we first implement a one-dimensional load-balancing algorithm for the master's sending operations, and then we test two variants for the workers to choose the next task to compute among those they have received: **FIFO** and Earliest Deadline First (**EDF**). That gives two heuristics for the online version of the algorithm (based on Section 5.4.3): **CBS3M**_EDF_ONLINE and **CBS3M**_FIFO_ONLINE). As a comparison basis, we also add two strategies, **CBS3M**_EDF_ROFF and **CBS3M**_FIFO_ROFF, with all information about future submissions: these strategies run the **CBS3M** algorithm under a rounded offline model: the algorithms have a complete information and compute the whole schedule at the beginning (based on Section 5.4.2), but are then adapted (rounded) to the one-port model.

Both the **CBS3M** and the **MWMA** strategies make use of linear programs to compute their schedules. These linear programs are solved using `glpk`, the Gnu Linear Programming Kit [71].

Experimental settings

In order to test and compare our heuristics, we perform both simulations and real experiments. Simulations are conducted using the SimGrid [94] simulator, while experiments make use of the MPICH-2 communication library [74].

Communication and computation times are realized by transmitting random data, or by computing random matrix products, as described below; no real application is used. In particular, this allows us to emulate a heterogeneous platform: we have full freedom to slow down some communications by transmitting the same data several times, and slow down some computations by performing several times the same task.

Our theoretical study is fully general, allowing computing times to be unrelated: a processor can process different applications with different speeds. However, for sake of simplicity, we consider in the experiments and in the simulations that we have uniform processors, and all applications are of the same kind: the processing time of a task depends only on its size (depending on the application) and the speed of the processor, not on the application. As we target a heterogeneous master-worker platform, we generate several platform scenarios. The computing speeds are uniformly distributed in interval $[\alpha, 10 \cdot \alpha]$, where α is the reference speed. Similarly, the link bandwidths are uniformly distributed in interval $[\beta, 10 \cdot \beta]$, where β is the reference bandwidth.

The experiments are conducted on a cluster composed of nine processors. The master is a SuperMicro server 6013PI, with a P4 Xeon 2.4 GHz processor, and the workers are all SuperMicro servers 5013-GM, with P4 2.4 GHz processors. All nodes have 1 GB of memory and are running Linux. They are connected with a switched 10 Mbps Fast Ethernet network.

Even if we totally control the platform parameters (computing speeds and bandwidths), when these characteristics are needed by a heuristic to take scheduling decisions, the parameters are measured within the program by sending a small message, or performing a small task. This is true both in the MPI implementation and in the simulations.

The time needed to measure the platform characteristics and take scheduling decisions is taken into account in the experiments (but not in the simulations). This phase usually takes a few seconds in the experiments (up to one minute) for scenarios of a few hours, and thus represents less than 1% of the total running time.

Applications

A bag-of-tasks application is described by its release date, its number of tasks, and the communication and computation sizes of one task. For our experiments and simulations, we randomly generated the applications, with the following constraints in order to be realistic:

- the release dates of the applications follow a log-normal distribution as suggested in [65];
- the total amount of communications and computations for an application is randomly chosen with a log-normal distribution between realistic bounds, and then split into tasks; the parameters used in the generation of the applications for the experiments and the simulations are described in Table 5.1.

The number of tasks for one application is upper-bounded by the minimum amount of communication and computation allowed for one task.

Results

In this section we describe the results obtained on all different platforms, experimental or simulated.

	parameter	experiments	simulations
general	number of workers	8	10
	number of applications	12	20
arrival dates	mean of the distribution in the log space	4.0	4.0
	standard deviation in the log space	1.2	1.2
computations	maximum amount of work application (Gflops)	76.8	409
	minimum amount of work per task (Gflops)	3.1	3.1
communications	maximum amount of communication per application (MB)	800	6,000
	minimum amount of communication per task (MB)	40	40
number of tasks	minimum number of tasks per application	10	20

Table 5.1: Parameters for the MPI experiments and for the SimGrid simulations.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
FIFO_RR	4.550	16.689	(± 7.897)	62.6	(the best in 0.0 %)
FIFO_MCT	1.857	6.912	(± 2.404)	17.9	(the best in 0.0 %)
FIFO_DD	4.550	16.689	(± 7.897)	62.6	(the best in 0.0 %)
NPSPT_RR	1.348	4.274	(± 1.771)	13.8	(the best in 0.0 %)
NPSPT_MCT	1.007	1.928	(± 0.610)	5.99	(the best in 1.3 %)
NPSPT_DD	1.348	4.274	(± 1.771)	13.8	(the best in 0.0 %)
SRPT_RR	1.348	4.121	(± 1.737)	13.8	(the best in 0.0 %)
SRPT_MCT	1.007	1.861	(± 0.601)	6.87	(the best in 2.2 %)
SRPT_DD	1.348	4.121	(± 1.737)	13.8	(the best in 0.0 %)
SWRPT_RR	1.344	4.119	(± 1.739)	13.8	(the best in 0.0 %)
SWRPT_MCT	1.007	1.857	(± 0.601)	6.87	(the best in 1.9 %)
SWRPT_DD	1.344	4.119	(± 1.739)	13.8	(the best in 0.0 %)
MWMA_NBT	1.477	3.433	(± 1.044)	8.49	(the best in 0.0 %)
MWMA_MS	2.435	8.619	(± 2.420)	20.4	(the best in 0.0 %)
CBS3M_FIFO_ONLINE	1.003	1.322	(± 0.208)	2.83	(the best in 6.9 %)
CBS3M_EDF_ONLINE	**1.003**	**1.163**	**(± 0.118)**	**1.93**	**(the best in 64.0 %)**
CBS3M_FIFO_ROFF	1.022	1.379	(± 0.276)	3.74	(the best in 3.8 %)
CBS3M_EDF_ROFF	**1.011**	**1.213**	**(± 0.125)**	**2.06**	**(the best in 26.2 %)**

Table 5.2: Simulation results: Relative max-stretch of all heuristics in the simulations.

Simulation results

In this section, we detail the results of the simulations. We run 1000 simulations based on the parameters described in Table 5.1. Table 5.2 presents the results of all heuristics for the max-stretch metric, whereas Figure 5.2 shows the evolution of some heuristics (the best ones) over the load of the scenario. Here the load is characterized with the optimal theoretical achievable max-stretch in the fluid model: we consider that a scenario where the optimal max-stretch is 6 is twice as loaded as a scenario with an optimal max-stretch of 3. All results are relative to the optimal max-stretch, which is computed in the offline case. A relative max-stretch of 1.5 means that the corresponding strategies achieves a max-stretch which is 1.5 times the optimal one, thus with a degradation of 50%.

The **CBS3M** heuristics perform very well for the max-stretch: **CBS3M**_EDF_ONLINE achieves the best max-stretch between all heuristics in 64% of the simulations. This heuristic performs significantly better than all other heuristics: it has an average max-stretch of 1.163 times the optimal max-stretch, the lowest standard deviation (0.118) and the minimum worst case (1.93) among all heuristics.

The good results of the **CBS3M** heuristics can be explained by the fact that they make very good use of the platform, by scheduling simultaneously several applications when it is possible,

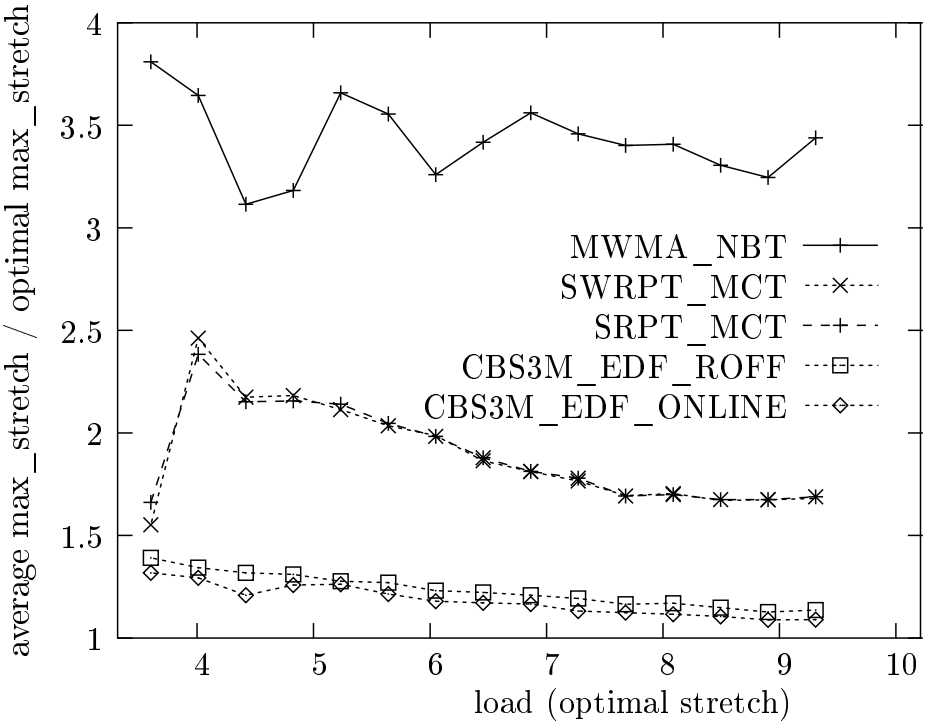


Figure 5.2: Simulation results: Evolution of the relative max-stretch of best heuristics in the simulations under different load conditions.

for example when the communication medium has still some free bandwidth after scheduling the most critical application. All other heuristics (except **MWMA**) are limited to scheduling only one application at a time, leading to an overall bad utilization of the computing platform. One can also note that the **CBS3M** heuristics also have very small standard deviations.

Another comment is the relative bad result of the involved strategies **MWMA_NBT** and **MWMA_MS**: although they schedule several applications concurrently on the platforms, they use a somewhat wrong computation of the priorities, leading to poor results.

In Figure 5.2, one can notice that, surprisingly, the rounded offline version of **CBS3M** is not always better than the online version. The offline version knows the future and thus should achieve better performance. However, it suffers from discrepancies between the actual characteristics of the platform and those of the platform model. The online version is able to circumvent this problem as it takes into account the work effectively processed to recompute the schedule at each new application arrival. This gain of reactivity compensates for the loss due to the lack of knowledge of the future. We also observe that resource selection is important on heterogeneous platforms, as the strategies which have the worst relative max-stretch are the ones using round-robin or demand-driven policies.

We also plot the results of the best heuristics for other objectives: sum-stretch (Table 5.3 and Figure 5.3), makespan (Table 5.4 and Figure 5.4), max-flow (Table 5.5 and Figure 5.5), and sum-flow(Table 5.6 and Figure 5.6). Quite surprisingly, **CBS3M** also gives the best average results for the makespan and the max-flow objectives. With respect to sum-flow, **CBS3M** gives the best results for light-loaded scenarios, whereas **SRPT** and **SWRPT** give better results for high-loaded scenarios. Finally, **CBS3M** is outperformed by **SRPT** and **SWRPT** for sum-stretch.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
FIFO_RR	2.064	6.783	(± 3.210)	30.7	(the best in 0.0 %)
FIFO_MCT	1.322	2.754	(± 0.670)	6.45	(the best in 0.0 %)
FIFO_DD	2.064	6.783	(± 3.210)	30.7	(the best in 0.0 %)
NPSPT_RR	1.019	2.942	(± 1.221)	10.1	(the best in 0.0 %)
NPSPT_MCT	1.000	1.182	(± 0.183)	2.53	(the best in 2.4 %)
NPSPT_DD	1.019	2.942	(± 1.221)	10.1	(the best in 0.0 %)
SRPT_RR	1.007	2.607	(± 1.071)	8.93	(the best in 0.0 %)
SRPT_MCT	**1.000**	**1.045**	**(± 0.098)**	**1.92**	**(the best in 25.5 %)**
SRPT_DD	1.007	2.607	(± 1.071)	8.93	(the best in 0.0 %)
SWRPT_RR	1.000	2.596	(± 1.068)	8.96	(the best in 0.1 %)
SWRPT_MCT	**1.000**	**1.038**	**(± 0.098)**	**1.92**	**(the best in 60.1 %)**
SWRPT_DD	1.000	2.596	(± 1.068)	8.96	(the best in 0.1 %)
MWMA_NBT	1.051	2.013	(± 0.644)	5.41	(the best in 0.0 %)
MWMA_MS	1.663	4.183	(± 1.269)	11.5	(the best in 0.0 %)
CBS3M_FIFO_ONLINE	1.000	1.294	(± 0.208)	2.16	(the best in 0.4 %)
CBS3M_EDF_ONLINE	**1.000**	**1.201**	**(± 0.190)**	**2.08**	**(the best in 20.2 %)**
CBS3M_FIFO_ROFF	1.000	1.332	(± 0.227)	2.57	(the best in 0.1 %)
CBS3M_EDF_ROFF	1.000	1.272	(± 0.214)	2.49	(the best in 3.8 %)

Table 5.3: Simulation results: Sum-stretch of all heuristics in the simulations.

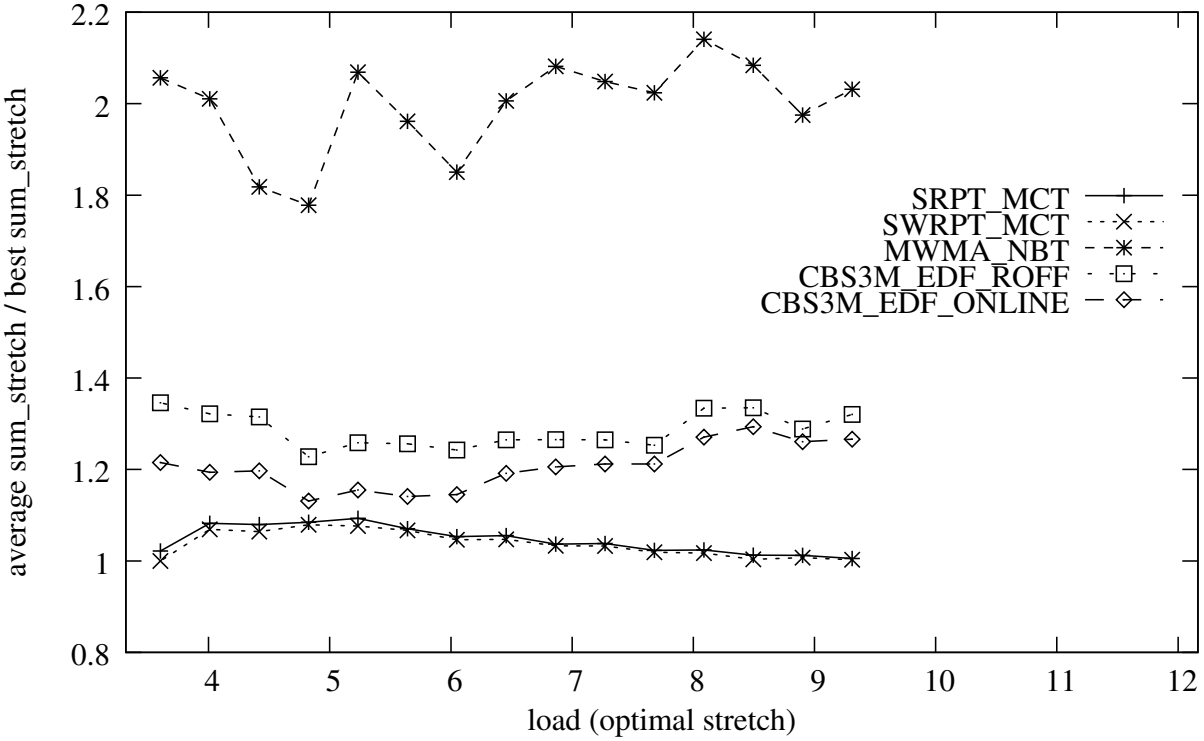


Figure 5.3: Simulation results: Evolution of the sum-stretch of best heuristics in the simulations under different load conditions.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
FIFO_RR	1.343	2.716	(± 0.684)	5.31	(the best in 0.0 %)
FIFO_MCT	1.000	1.329	(± 0.202)	2.11	(the best in 0.1 %)
FIFO_DD	1.343	2.716	(± 0.684)	5.31	(the best in 0.0 %)
NPSPT_RR	1.325	2.714	(± 0.685)	5.33	(the best in 0.0 %)
NPSPT_MCT	1.000	1.329	(± 0.202)	2.1	(the best in 0.0 %)
NPSPT_DD	1.325	2.714	(± 0.685)	5.33	(the best in 0.0 %)
SRPT_RR	1.325	2.714	(± 0.686)	5.32	(the best in 0.0 %)
SRPT_MCT	1.000	1.328	(± 0.202)	2.1	(the best in 0.0 %)
SRPT_DD	1.325	2.714	(± 0.686)	5.32	(the best in 0.0 %)
SWRPT_RR	1.322	2.715	(± 0.686)	5.32	(the best in 0.0 %)
SWRPT_MCT	1.000	1.328	(± 0.202)	2.1	(the best in 0.0 %)
SWRPT_DD	1.322	2.715	(± 0.686)	5.32	(the best in 0.0 %)
MWMA_NBT	1.000	1.079	(± 0.070)	1.45	(the best in 4.6 %)
MWMA_MS	1.000	1.078	(± 0.067)	1.42	(the best in 2.1 %)
CBS3M_FIFO_ONLINE	1.000	1.029	(± 0.029)	1.17	(the best in 7.5 %)
CBS3M_EDF_ONLINE	**1.000**	**1.004**	**(± 0.006)**	**1.05**	**(the best in 35.0 %)**
CBS3M_FIFO_ROFF	**1.000**	**1.018**	**(± 0.023)**	**1.22**	**(the best in 17.6 %)**
CBS3M_EDF_ROFF	**1.000**	**1.003**	**(± 0.006)**	**1.07**	**(the best in 53.0 %)**

Table 5.4: Simulation results: Makespan of all heuristics in the simulations.

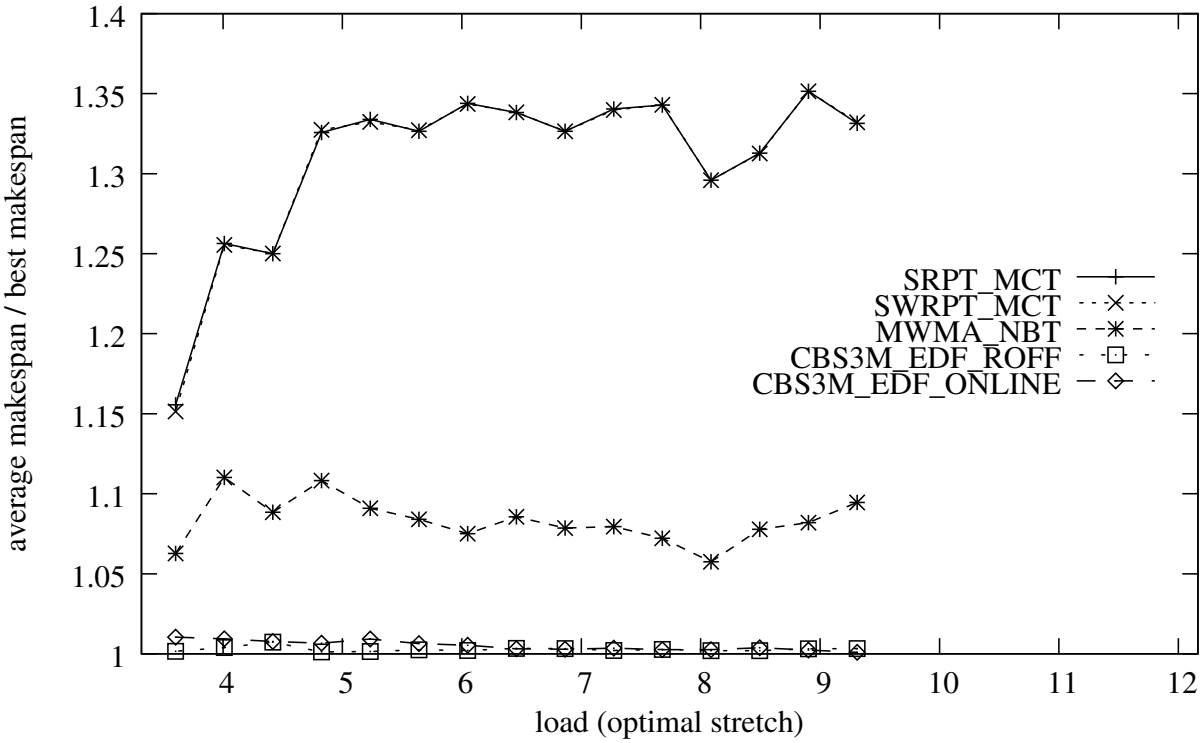


Figure 5.4: Simulation results: Evolution of the makespan of best heuristics in the simulations under different load conditions.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
FIFO_RR	1.146	3.097	(± 1.135)	10.2	(the best in 0.0 %)
FIFO_MCT	**1.000**	**1.281**	**(± 0.258)**	**2.83**	**(the best in 14.4 %)**
FIFO_DD	1.146	3.097	(± 1.135)	10.2	(the best in 0.0 %)
NPSPT_RR	1.386	3.282	(± 1.222)	10.9	(the best in 0.0 %)
NPSPT_MCT	1.002	1.460	(± 0.287)	3.09	(the best in 0.0 %)
NPSPT_DD	1.386	3.282	(± 1.222)	10.9	(the best in 0.0 %)
SRPT_RR	1.386	3.289	(± 1.225)	10.9	(the best in 0.0 %)
SRPT_MCT	1.003	1.473	(± 0.306)	4.28	(the best in 0.0 %)
SRPT_DD	1.386	3.289	(± 1.225)	10.9	(the best in 0.0 %)
SWRPT_RR	1.382	3.291	(± 1.225)	10.9	(the best in 0.0 %)
SWRPT_MCT	1.000	1.477	(± 0.309)	4.28	(the best in 0.1 %)
SWRPT_DD	1.382	3.291	(± 1.225)	10.9	(the best in 0.0 %)
MWMA_NBT	1.000	1.181	(± 0.153)	1.99	(the best in 7.0 %)
MWMA_MS	1.000	1.261	(± 0.189)	2.32	(the best in 1.1 %)
CBS3M_FIFO_ONLINE	1.000	1.054	(± 0.061)	1.52	(the best in 5.8 %)
CBS3M_EDF_ONLINE	**1.000**	**1.031**	**(± 0.057)**	**1.48**	**(the best in 23.2 %)**
CBS3M_FIFO_ROFF	**1.000**	**1.037**	**(± 0.058)**	**1.48**	**(the best in 21.6 %)**
CBS3M_EDF_ROFF	**1.000**	**1.023**	**(± 0.055)**	**1.48**	**(the best in 48.7 %)**

Table 5.5: Simulation results: Max-flow of all heuristics in the simulations.

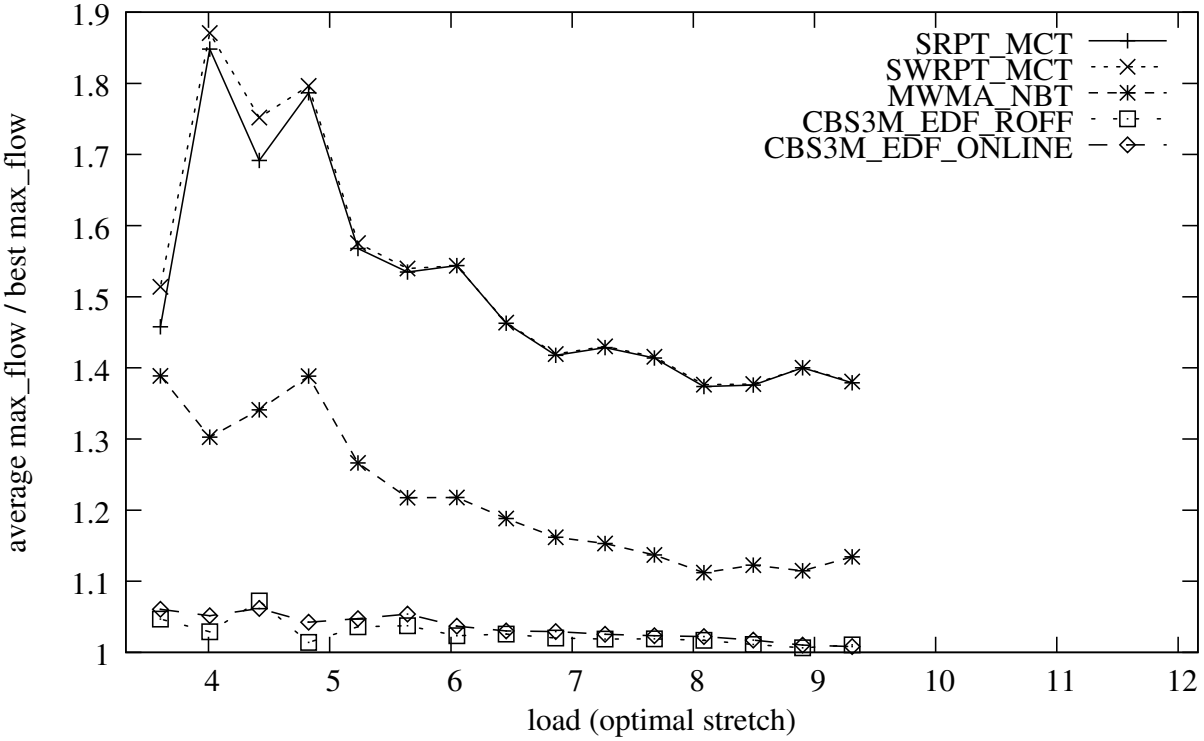


Figure 5.5: Simulation results: Evolution of the max-flow of best heuristics in the simulations under different load conditions.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
FIFO_RR	1.644	4.020	(± 1.567)	16.3	(the best in 0.0 %)
FIFO_MCT	1.134	1.652	(± 0.264)	3.33	(the best in 0.0 %)
FIFO_DD	1.644	4.020	(± 1.567)	16.3	(the best in 0.0 %)
NPSPT_RR	1.196	2.811	(± 1.081)	9.21	(the best in 0.0 %)
NPSPT_MCT	1.000	1.149	(± 0.171)	2.32	(the best in 3.5 %)
NPSPT_DD	1.196	2.811	(± 1.081)	9.21	(the best in 0.0 %)
SRPT_RR	1.079	2.704	(± 1.048)	9.03	(the best in 0.0 %)
SRPT_MCT	**1.000**	**1.105**	**(± 0.151)**	**2.23**	**(the best in 32.1 %)**
SRPT_DD	1.079	2.704	(± 1.048)	9.03	(the best in 0.0 %)
SWRPT_RR	1.079	2.706	(± 1.049)	9.03	(the best in 0.0 %)
SWRPT_MCT	**1.000**	**1.108**	**(± 0.152)**	**2.23**	**(the best in 15.4 %)**
SWRPT_DD	1.079	2.706	(± 1.049)	9.03	(the best in 0.0 %)
MWMA_NBT	1.000	1.404	(± 0.217)	2.29	(the best in 0.1 %)
MWMA_MS	1.359	2.333	(± 0.355)	3.7	(the best in 0.0 %)
CBS3M_FIFO_ONLINE	1.000	1.122	(± 0.101)	1.62	(the best in 1.4 %)
CBS3M_EDF_ONLINE	**1.000**	**1.065**	**(± 0.090)**	**1.53**	**(the best in 35.6 %)**
CBS3M_FIFO_ROFF	1.000	1.120	(± 0.103)	1.67	(the best in 0.3 %)
CBS3M_EDF_ROFF	**1.000**	**1.087**	**(± 0.101)**	**1.66**	**(the best in 18.7 %)**

Table 5.6: Simulation results: Sum-flow of all heuristics in the simulations.

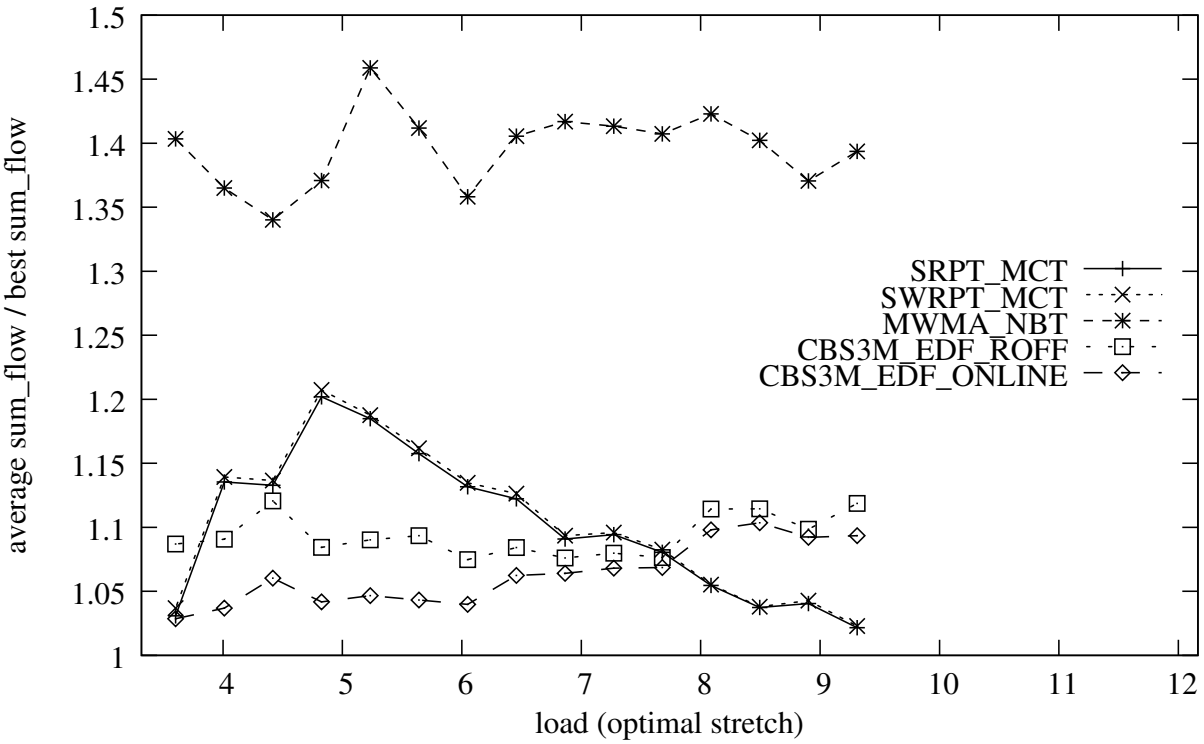


Figure 5.6: Simulation results: Evolution of the sum-flow of best heuristics in the simulations under different load conditions.

Experimental results

We now move to the real experiments with MPI communications. The experiments were performed on 50 different platform and application settings. As several heuristics performed very poorly in the simulations, especially the heuristics based on round-robin and demand-driven policies, and thus would have lead to huge computation times, we discarded them and restricted ourselves to a smaller set of heuristics in order to get reasonable running times. Once again, the performance of a given strategy is measured through its relative max-stretch, that is the ratio between the obtained max-stretch and the theoretical optimal max-stretch in the fluid model.

Algorithm	minimum	average	(± stddev)	maximum	(fraction of best result)
CBS3M_EDF_ROFF	**1.04**	**1.30**	**(± 0.13)**	**1.63**	**(the best in 38.0%)**
CBS3M_EDF_ONLINE	**1.02**	**1.41**	**(± 0.30)**	**2.09**	**(the best in 30.0%)**
CBS3M_FIFO_ROFF	1.04	1.38	(± 0.28)	2.97	(the best in 12.0%)
CBS3M_FIFO_ONLINE	1.02	1.46	(± 0.26)	1.96	(the best in 6.0%)
FIFO_MCT	1.10	1.81	(± 0.60)	4.15	(the best in 4.0%)
FIFO_RR	1.35	4.99	(± 3.46)	19.50	(the best in 0.0%)
MWMA_MS	1.22	2.29	(± 0.56)	4.05	(the best in 0.0%)
MWMA_NBT	1.13	1.50	(± 0.17)	2.06	(the best in 4.0%)
NPSPT_DD	1.33	4.87	(± 3.10)	18.75	(the best in 0.0%)
NPSPT_MCT	1.08	1.84	(± 0.61)	3.43	(the best in 4.0%)
SRPT_MCT	1.09	1.87	(± 0.59)	3.38	(the best in 0.0%)
SWRPT_MCT	1.08	1.88	(± 0.59)	3.38	(the best in 2.0%)

Table 5.7: MPI experiment results: Relative max-stretch of selected heuristics in the experiments.

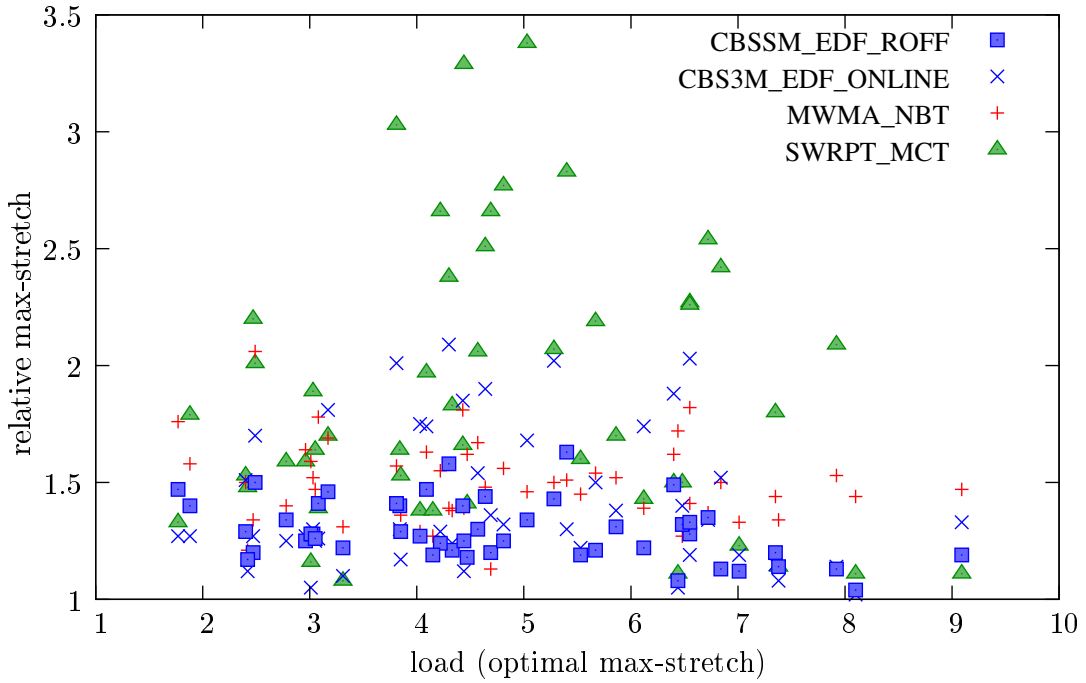


Figure 5.7: MPI experiment results: Evolution of the relative max-stretch of best heuristics in the simulations under different load conditions.

The results of the experiments are summarized in Table 5.7; Figure 5.7 presents the results for the best four strategies: **CBS3M** using EDF policy, in both the offline and online versions, **MWMA_NBT** and **SWRPT**.

These results are quite similar to the simulation results. We can see in Table 5.7 that the four versions of **CBS3M** achieve a better relative max-stretch than most other strategies. In fact, they all achieve far better performance than any other strategy in 86% of the experiments. Once again the online version performs generally better than the offline version, as explained earlier.

The major difference concerns the **MWMA** strategies, which perform much better than in the simulations. The **MWMA_NBT** algorithm lies in between our algorithms and the greedy strategies, even if sometimes they achieve a very bad relative max-stretch (up to 2.06). This can be explained by the different scenarios used in experiments and simulations: in order to avoid huge running times in the experiments, we concentrate on simpler scenarios, with smaller applications, whereas in the simulations, we consider larger applications as simulations run for a short time even with long simulated running times. To fully assess the adequacy of the simulations and the experiments, we decided to re-run the experimental scenarios within our simulator, and to compare both results.

Simulations of experimental settings

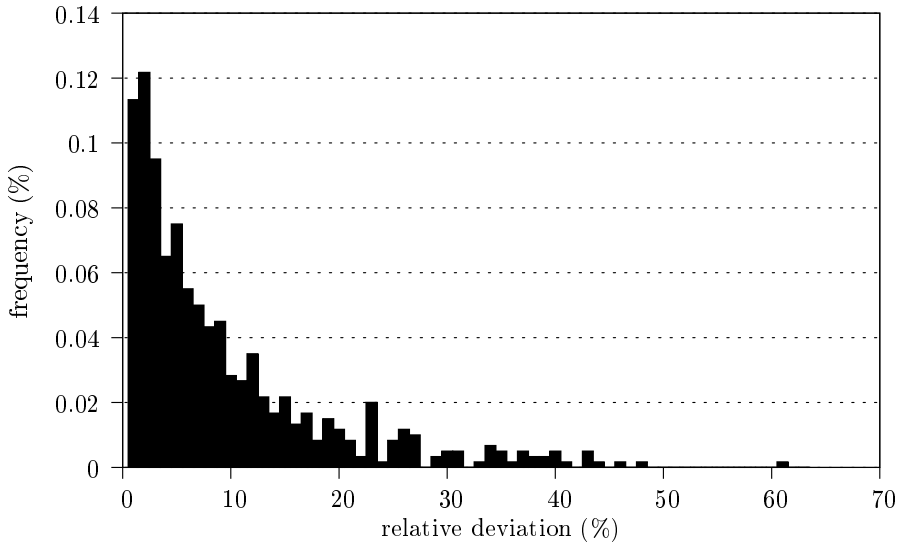


Figure 5.8: Distribution of relative deviation between simulations and experiments.

In this section we check the accuracy of our simulations by "simulating the experiments": we run simulations on the same scenarios (platforms and application parameters) that have been used for real experiments. Obviously, the executions will differ, and we do not expect the results to be strictly identical: the simulations do not account for the dynamic nature of the platform used in real experiments. Simulations do not take scheduling times into account and rely on exact application/platform parameters, while experiments can only rely on inaccurate predicted values.

In Figure 5.8, we plot the distribution of the relative deviation between the max-stretch obtained in the experiments and the max-stretch obtained in the simulations, for all strategies. The maximum deviation is 60.1%, but the average deviation is only 8.9%, with a standard deviation of 9.5% (the median value is 5.5%). Overall, the accuracy of the simulations is satisfactory, and even good if we keep in mind all possible sources of differences between simulations and experiments. Furthermore, the difference between the **CBS3M** heuristics and the other algorithms were most of the time above 20%.

5.4.5 Related work

Related literature can be classified into two main categories: steady-state scheduling and flow-type objective functions.

Steady-State Scheduling

Minimizing the makespan, i.e., the total execution time, is an NP-hard problem in most practical situations [68, 130, 51], while it turns out that the optimal steady-state schedule can often be characterized very efficiently, with low-degree polynomial complexity.

The steady-state approach has been pioneered by Bertsimas and Gamarnik [28]. It has been used successfully in many situations [21]. In particular, steady-state scheduling has been used to schedule independent tasks on heterogeneous tree-overlay networks [14, 9]. Bandwidth-centric scheduling is introduced in [14], and extensive experiments are reported in [90]. The steady-state approach has also been used by Hong et al. [79] who extend the work of [14] to deploy a divisible workload on a heterogeneous platform. However, and to the best of our knowledge, the only reference dealing with steady-state scheduling for several applications is [17].

Flow-type objective functions and online scheduling

Most of the existing work on stretch minimization deals with the mono-processor case. In fact, there has been a lot of work on the performance of simple list scheduling heuristics for the optimization of flow-like metrics with preemption. We will therefore first consider this work.

Flow optimization. On a single processor, the max-flow is optimized by *First-Come First-Serve* (FCFS) (see Bender et al. [24] for example), and the sum-flow is optimized by *shortest remaining processing time first* (SRPT) [6].

Things are more difficult for stretch minimization. First, any online algorithm which has a better competitive ratio for sum-stretch minimization than FCFS is subject to starvation, and is thus not a competitive algorithm for max-stretch minimization [97]. In other words, the two objective functions cannot be optimized simultaneously to obtain a non trivial competitive factor (FCFS is not taking into account the weight of tasks in the objective).

Sum-stretch minimization. The complexity of the offline minimization of the sum-stretch with preemption is still an open problem. At the very least, this is a hint at the difficulty of this problem. Bender, Muthukrishnan, and Rajaraman [25] designed a Polynomial Time Approximation Scheme (PTAS) for minimizing the sum-stretch with preemption. Chekuri and Khanna [47] proposed an approximation scheme for the more general sum weighted flow minimization problem. On the online side, no online algorithm has a competitive ratio less than or equal to 1.19484 for the minimization of sum-stretch [96, 97].

As we recalled, on one processor, SRPT is optimal for minimizing the sum-flow. When SRPT takes a scheduling decision, it only considers the *remaining* processing time of a task, and not its *original* processing time, i.e., the *weight* of the task in the objective function. Nevertheless, Muthukrishnan, Rajaraman, Shaheen, and Gehrke have shown [106] that SRPT is 2-competitive for sum-stretch. Another well studied algorithm is the Smith's ratio rule [137] also known as *shortest weighted processing time* (SWPT). Whatever the weights, SWPT is 2-competitive [125] for the minimization of the sum of weighted completion times. However, SWPT is not an approximation algorithm for minimizing the sum-stretch. Indeed, both SPT (*shortest processing time*) and SWPT are not competitive algorithms for minimizing the sum-stretch [96, 97]. To address the weaknesses of both SRPT and SWPT, one might consider a heuristic that takes into account both the original and the remaining processing times of the jobs, which leads to the *shortest weighted remaining processing time* heuristic (SWRPT). Muthukrishnan, Rajaraman, Shaheen, and Gehrke [106] proved that SWRPT is actually optimal when there are only two job sizes. However, in the general case, the worst case for SWRPT for sum-stretch minimization is at least 2, and thus is no better than that of SRPT [96, 97].

Max-stretch minimization. Max-stretch can be optimally minimized in the offline case [96, 97], even on unrelated machines (either with preemption or in the divisible load framework). The online case is far more difficult. With only two task sizes, SWRPT is optimal, as we have already recalled. However, as soon as there are at least three task sizes, no algorithm has a competitive ratio lower than $\frac{1}{2}\Delta^{\sqrt{2}-1}$, where Δ is the ratio of the largest to the smallest size of tasks [96, 97].

In fact, this latter work is the only one targeting max stretch minimization in a multiprocessor environment. This work is done in the divisible load framework, meaning that applications can be arbitrarily divided in sub-tasks when, in the context of the current paper, the granularity of the tasks of each application is fixed independently of the scheduler. Furthermore, communications can be neglected for the applications targeted in [96, 97], when they play a major role in our case.

General online scheduling. More generally, we refer the reader to surveys on online scheduling algorithms [118], on randomized online scheduling algorithms [2], or even more generally on online algorithms [3].

5.5 Minimizing power consumption

In this second part, we only consider a single fully parallel application $\mathcal{A}$ but we suppose that each processor has several possible speeds of computation (or modes). This context introduces some new problems related to power consumption. For example, one may want to decrease the operating cost of clusters. The Earth Simulator requires about 12 megawatts of peak power, and Petaflop systems may require 100 MegaWatts of power, nearly the output of a small power plant (300 MegaWatts). At \$100 per MegaWatt, peak operation of this petaflop machine is \$10,000 per hour [69]. And these estimates ignore the additional cost of dedicated cooling. Even without considering battery-powered systems such as laptops and embedded systems, the consumption is a critical factor as most of the power consumed is released as heat by the processors. And the heat generated in supercomputers becomes harder and harder to dissipate. Current estimations state that cooling solutions are rising at \$1 to \$3 per watt of heat dissipated [135].

Under such assumptions, one can understand the need to focus on minimizing power consumption. We first present different power consumption models before the study of our problem at the processor level, and its extension to the system level. At the end, we introduce a more accurate (but also more intricate) power consumption model.

5.5.1 Models

Two main system-level energy-saving techniques are Dynamic Voltage Scaling (DVS) —which enables to select a processor's supply voltage according to task requirements— and Power Management —which enables to shut down a processor when idle. Among them, DVS is recognized as one of the most effective power-reduction techniques, and it is now present on mobile platforms and even on high-performance microprocessors.

Dynamic Voltage Scaling

For processors based on CMOS technology, the power consumption is dominated by the dynamic power dissipation P_d, which is given as a function of the operating frequency:

$$P_d = C_{\text{eff}} \cdot V^2 \cdot s$$

where C_{eff} is the average switched capacitance per cycle, V is the operating voltage, and s is the operating frequency.

DVS works on a very simple principle: decreasing the supply voltage to the CPU consumes less power. But s and V are not independent; there is a minimum voltage required to drive the microprocessor at the desired frequency. In fact, there is a linear relation between the frequency and the minimum voltage of the processor. So DVS reduces the power consumption by changing the clock frequency-voltage settings. For this reason, DVS is also called *frequency-scaling* or *speed scaling*.

Most authors use as power consumption law the expression $P_d = s^\alpha$, where $\alpha > 1$. During this work, we will take a more general approach, as our results do not assume a specific relationship between speed and power. We will denote by $\mathfrak{P}_u$ the power consumption per time unit of the processor P_u. We only assume that power consumption is a continuous, strictly-convex function of the processor's speed. Basically, our assumption is that the slower a task is executed, the less energy is used to complete the task, which is clearly a realistic hypothesis. Formally,

that means that the line segment between any two points on the power/speed curve lies above the curve, or that the power increases super-linearly with speed.

Definition 5.1. *(super-linearity) F is a function super-linear if F is increasing and convex.*

So the function $P_d(s)$, which gives the power consumption according to the speed s of the processor, has the following property (one can note that P_d may only be defined between the extremum values of the processor speeds):

$$\forall\ 0 \leq s < s_{\max},\ 0 < \lambda_1 < s_{\max} - s,\ 0 < \lambda_2 \leq s_{\max} - s - \lambda_1,$$
$$\frac{P_d(s+\lambda_1) - P_d(s)}{\lambda_1} \leq \frac{P_d(s+\lambda_1+\lambda_2) - P_d(s+\lambda_1)}{\lambda_2}.$$

Multi-mode processors

In order to be more realistic, we suppose that we have a discrete voltage-scaling model. The computational speed of worker P_u has to be picked among a limited number of m_u modes. We denote the computational speeds $s_{u,i}$, meaning that the processor P_u running in the ith mode takes $X/s_{u,i}$ time-units to execute X floating point operations. We introduce a new virtual processor $P_{u,i}$ that represents the processor P_u running in the ith mode. The power consumption per time-unit of $P_{u,i}$ is denoted by $\mathfrak{P}_{u,i}$. We will suppose that the processing speeds are listed in increasing order on each processor ($s_{u,1} \leq s_{u,2} \leq \cdots \leq s_{u,m_u}$). One can note that the time required to process one task of $\mathcal{A}$ of size w on virtual processor $P_{u,i}$ is thus $w/s_{u,i}$. Of course, the modes are exclusive; one processor can only run at a single mode at any given time.

Under an ideal model, we suppose that switching among the modes does not cost any penalty. In real life, it costs a penalty depending on the modes. There exist two kinds of overhead that have to be considered when changing the processor speed: the time overhead and the power overhead.

Timing overhead can be represented as

$$T_{overhead} = C_T + K_T|s_1 - s_2|,$$

where C_T and K_T are constants, and s_1 and s_2 being respectively the processor speeds before and after the adjustment. Most of the time, the authors suppose that $T_{overhead} = 0$, since processors can still execute instructions during transitions [40] and timing overhead is linear to the speed of the processor.

Power overhead is very similar:

$$P_{overhead} = C_P + K_P|V_1^2 - V_2^2|,$$

where C_P and K_P are still constants, and V_1 and V_2 being respectively the processor voltages before and after the adjustment. This overhead is more important, as it depends on the *square* of the voltages. If $K_P \approx 0$, we suppose a constant consumption overhead.

Of course, the variables C_1 and C_P depend on the processor, so we need to introduce the notations $C_1^{(u)}$ and $C_P^{(u)}$.

We may also wonder what happens when the utilization of a processor tends to zero. There also exist two policies:

- **ideal model:** in this model an idle processor does not consume any power, so the power consumption is super-linear from 0 to the power consumption at frequency $s_{u,1}$;
- **idle consumption model:** once a processor is on, it will always be above a minimal power consumption defined by its idle frequency/speed $s_{u,1}$.

Bicriteria

As defined in the introduction, our goal is bi-criteria. The first objective is to minimize the power consumption, while assuming an ideal model, and the second objective is the maximization of the throughput.

We denote by $\rho_{u,i}$ the throughput of worker P_u under mode $m_{u,i}$ for application $\mathcal{A}$, i.e., the average number of tasks of $\mathcal{A}$ that $P_{u,i}$ executes each time-unit. Of course, there is a limit to the number of tasks that each virtual processor can perform per time-unit. First of all, as $P_{u,i}$ runs at speed $s_{u,i}$, it can not have a throughput greater than $w/s_{u,i}$ tasks per time-unit. Second, as all modes of P_u are exclusive, if $P_{u,i}$ is at its maximal throughput, no other virtual processor can be computing. So their is a strong relationship between the throughput of one mode and the maximum throughput available for all remaining modes. As $\frac{\rho_{u,i}\ w}{s_{u,i}}$ represents the fraction of time spent under mode $m_{u,i}$ per time-unit, this constraint can be expressed by:

$$\sum_{i=1}^{m_u} \frac{\rho_{u,i}\ w}{s_{u,i}} \leq 1$$

As we are in the ideal model and for simplicity of the proofs in the next section, we will add an additional idle mode $P_{u,0}$ whose speed is $s_{u,0} = 0$.

The power consumption per time-unit of $P_{u,i}$, when fully used, is $\mathfrak{P}_{u,i}$ (to be coherent, we have $\mathfrak{P}_{u,0} = 0$). Its power consumption per time-unit with a throughput of $\rho_{u,i}$ is then $\frac{\rho_{u,i}\ w}{s_{u,i}}\mathfrak{P}_{u,i} = \rho_{u,i}\mathfrak{K}_{u,i}$. Thanks to the super-linearity of $\mathfrak{P}$, $\mathfrak{K}_{u,i}$ is increasing with i and convex.

We still denote by ρ_u the throughput of worker P_u, i.e., the sum of the throughput of each virtual processor of P_u (except the throughput of the idle mode), so the total throughput is denoted by:

$$\rho = \sum_{u=1}^{p} \rho_u = \sum_{u=1}^{p} \sum_{i=1}^{m_u} \rho_{u,i}.$$

We assume in the next sections (Section 5.5.2 and 5.5.3) that the power consumption is without power and timing overhead and ideal (a processor can be turned off without any cost). We will extend the work to a more realistic framework in section 5.5.4.

5.5.2 At the processor level

In this section, we look at both problems at the processor level: maximizing the throughput given an upper bound on power consumption and minimizing the power consumption given a lower bound on throughput. We no longer consider the master, and the communication links, but focus on the processor version of the problems. Building on previous works, we will find the best way to choose between the processor modes and to determine the minimum power consumption achievable. This will lead us to closed formulas linking the power consumption of processors to their throughput. These formulas will be very useful when dealing with the

multi-processor problem in Section 5.5.3.

The processor-level simpler problems have often been visited by previous works. For a single processor, [83] already proved that the voltages minimizing power consumption on discrete models are the immediate neighbors of the optimal voltage in the continuous model (where the voltage of a processor can take any arbitrary value) [83], however, gave neither the percentage of utilization of each mode, nor the power consumed according to the optimal voltage. The authors of [83] also proved these results for a specific relationship between the modes and the power consumption, whereas we show that it can be generalized. At last, these authors did not look at the problem of maximizing the throughput given a power consumption bound.

Power consumption minimization

The minimization of the power consumption is bounded by two types of constraints:

- The first constraint states that the processor has to ensure a given throughput,
- The second constraint states that the processing capacity of $P_{u,i}$ can not be exceeded, and that the different modes are exclusive.

So our optimization problem is:

$$\begin{cases} \text{MINIMIZE } \mathfrak{P}_u = \sum_{i=1}^{m_u} \rho_{u,i}\mathfrak{K}_{u,i} \text{ SUBJECT TO} \\ \sum_{i=1}^{m_u} \rho_{u,i} = \rho_u \\ \sum_{i=1}^{m_u} \frac{\rho_{u,i}}{s_{u,i}} w \ \leq \ 1 \end{cases} \tag{5.21}$$

A first remark is that the throughput that the processor has to achieve must be lower than its maximum throughput ($\rho_u \leq \frac{s_{u,m_u}}{w}$), otherwise the system has no solution.

Linear program (5.21) can easily be solved over the rationals, and the throughput of the modes of the processor depend on the total throughput that has to be achieved.

Lemma 5.1. *If $\rho_u > 0$ and $\exists\ i_0,\ 0 \leq i_0 < m_u,\ \frac{s_{u,i_0}}{w} < \rho_u \leq \frac{s_{u,i_0+1}}{w}$, then an optimal solution for power consumption minimization is:*

$$\tilde{\rho}_{u,i_0} = \frac{s_{u,i_0}(s_{u,i_0+1} - w\rho_u)}{w(s_{u,i_0+1} - s_{u,i_0})} \qquad \tilde{\rho}_{u,i_0+1} = \frac{s_{u,i_0+1}(w\rho_u - s_{u,i_0})}{w(s_{u,i_0+1} - s_{u,i_0})} \qquad \tilde{\rho}_{u,i} = 0 \text{ if } i \notin \{i_0, i_0+1\}$$

Otherwise, the processor cannot achieve the given throughput.

Proof. We first show that $\tilde{\boldsymbol{\mathcal{S}}}$ **is feasible:**

$$\begin{aligned} \sum_{i=1}^{m_u} \tilde{\rho}_{u,i} &= \frac{s_{u,i_0}(s_{u,i_0+1} - w\rho_u) + s_{u,i_0+1}(w\rho_u - s_{u,i_0})}{w(s_{u,i_0+1} - s_{u,i_0})} = \rho_u; \\ \sum_{i=1}^{m_u} \frac{\tilde{\rho}_{u,i} w}{s_{u,i}} &= \frac{(s_{u,i_0+1} - w\rho_u)}{s_{u,i_0+1} - s_{u,i_0}} + \frac{(w\rho_u - s_{u,i_0})}{s_{u,i_0+1} - s_{u,i_0}} = 1. \end{aligned}$$

Let $\mathcal{S}'$ be an optimal solution, $\mathcal{S}' = \{\rho'_{u,1}, \cdots, \rho'_{u,m_u}\}$. As $\mathcal{S}'$ is a solution, it respects all the constraints of Linear program (5.21). So:

$$\sum_{i=1}^{m_u} \rho'_{u,i} = \rho_u \quad \text{and} \quad \sum_{i=1}^{m_u} \frac{\rho'_{u,i} w}{s_{u,i}} \leq 1.$$

Let $i_{\min}$ be the slowest mode used by $\mathcal{S}'$, and $i_{\max}$ the fastest. Then we can distinguish three cases:

- **If $\boldsymbol{i_{\min} > i_0}$ or $\boldsymbol{i_{\min} = i_0}$ and $\boldsymbol{\rho'_{u,i_0} < \tilde{\rho}_{u,i_0}}$:** In both cases, $\rho'_{u,i_0} < \tilde{\rho}_{u,i_0}$, so there exists $\epsilon > 0$, such that $\rho'_{u,i_0} = \tilde{\rho}_{u,i_0} - \epsilon$. Then we can look at the power consumption of $\mathcal{S}'$:

$$\begin{aligned}
\sum_{i=1}^{m_u} \rho'_{u,i} \mathfrak{K}_{u,i} &\geq \rho'_{u,i_0} \mathfrak{K}_{u,i_0} + \left(\sum_{i=i_0+1}^{m_u} \rho'_{u,i} \right) \mathfrak{K}_{u,i_0+1} \\
&= \rho'_{u,i_0} \mathfrak{K}_{u,i_0} + (\rho_u - \rho'_{u,i_0}) \mathfrak{K}_{u,i_0+1} \\
&= (\tilde{\rho}_{u,i_0} - \epsilon) \mathfrak{K}_{u,i_0} + (\rho_u - \tilde{\rho}_{u,i_0} + \epsilon) \mathfrak{K}_{u,i_0+1} \\
&= \tilde{\rho}_{u,i_0} \mathfrak{K}_{u,i_0} + \tilde{\rho}_{u,i_0+1} \mathfrak{K}_{u,i_0+1} + \epsilon \left(\mathfrak{K}_{u,i_0+1} - \mathfrak{K}_{u,i_0} \right) \\
&\geq \tilde{\rho}_{u,i_0} \mathfrak{K}_{u,i_0} + \tilde{\rho}_{u,i_0+1} \mathfrak{K}_{u,i_0+1}.
\end{aligned}$$

  And so our solution does not consume more power, and is thus also optimal.

- **If $\boldsymbol{i_{\max} < i_0 + 1}$ or $\boldsymbol{i_{\max} = i_0 + 1}$ and $\boldsymbol{\rho'_{u,i_0+1} < \tilde{\rho}_{u,i_0+1}}$:** In both cases, $\rho'_{u,i_0+1} < \tilde{\rho}_{u,i_0+1}$, so there exists $\epsilon > 0$, such that $\rho'_{u,i_0+1} = \tilde{\rho}_{u,i_0+1} - \epsilon$. Then, we have:

$$\sum_{i=1}^{i_0} \rho'_{u,i} = \rho_u - \rho'_{u,i_0+1} \geq \rho_u - \tilde{\rho}_{u,i_0+1} + \epsilon = \tilde{\rho}_{u,i_0} + \epsilon$$

$$\begin{aligned}
\text{And} \quad \sum_{i=1}^{i_{\max}} \frac{\rho'_{u,i} w}{s_{u,i}} &= \sum_{i=1}^{i_0} \frac{\rho'_{u,i} w}{s_{u,i}} + \frac{\rho'_{u,i_0+1} w}{s_{u,i_0+1}} \\
&\geq \frac{w \sum_{i=1}^{i_0} \rho'_{u,i}}{s_{u,i_0}} + \frac{\rho'_{u,i_0+1} w}{s_{u,i_0+1}} \\
&\geq w \frac{\tilde{\rho}_{u,i_0} + \epsilon}{s_{u,i_0}} + w \frac{\tilde{\rho}_{u,i_0+1} - \epsilon}{s_{u,i_0+1}} \geq 1 + w\epsilon \left(\frac{1}{s_{u,i_0}} - \frac{1}{s_{u,i_0+1}} \right) > 1.
\end{aligned}$$

  which is in contradiction with the second constraint.

- **Otherwise** we know that either $i_{\min} < i_0$, so $\rho'_{u,i_{\min}} \geq \tilde{\rho}_{u,i_{\min}} = 0$, or $i_{\min} = i_0$ and $\rho'_{u,i_{\min}} \geq \tilde{\rho}_{u,i_{\min}}$. In both cases $\rho'_{u,i_{\min}} \geq \tilde{\rho}_{u,i_{\min}}$, and, for the same reasons, $\rho'_{u,i_{\max}} \geq \tilde{\rho}_{u,i_{\max}}$. We also know that (at least) one virtual processor among P_{u,i_0} and P_{u,i_0+1} has a throughput in $\mathcal{S}'$ strictly smaller than in $\tilde{\mathcal{S}}$ (otherwise the power consumption of $\mathcal{S}'$ is greater). Let call that processor P_α. The idea of the proof is to give an amount $\epsilon_{\min}$ of the work of $P_{i_{\min}}$ to P_α. As P_α is faster than $P_{i_{\min}}$, it takes less time to P_α to process $\epsilon_{\min}$ than to $P_{i_{\min}}$. During the spared time, P_α has time to do an amount $\epsilon_{\max}$ of the work of $P_{i_{\max}}$. Basically, $\epsilon_{\min}$ and $\epsilon_{\max}$ are defined such as the throughput in the new scheduling $\mathcal{S}''$ of either $P_{i_{\min}}$, or $P_{i_{\max}}$ is set to its throughput in $\tilde{\mathcal{S}}$:

$$\begin{cases}
\rho''_{u,i_{\min}} &= \rho'_{u,i_{\min}} - \epsilon_{\min} \\
\rho''_{u,\alpha} &= \rho'_{u,\alpha} + \epsilon_{\min} + \epsilon_{\max} \\
\rho''_{u,i_{\max}} &= \rho'_{u,i_{\max}} - \epsilon_{\max} \\
\rho''_{u,i} &= \rho'_{u,i} \quad \text{otherwise.}
\end{cases}$$

$\epsilon_{\min} = \min\left\{\rho'_{u,i_{\min}} - \tilde{\rho}_{u,i_{\min}}; \frac{(\rho'_{u,i_{\max}} - \tilde{\rho}_{u,i_{\max}})}{\lambda}\right\}, \epsilon_{\max} = \epsilon_{\min}\lambda, \lambda = \frac{s_{u,i_{\max}}(s_{u,\alpha} - s_{u,i_{\min}})}{s_{u,i_{\min}}(s_{u,i_{\max}} - s_{u,\alpha})}$.
λ gives the relation between the amount of work taken from $P_{i_{\min}}$ and the amount of work of $P_{i_{\max}}$ that can be performed by P_α during its spared time.

$\mathcal{S}''$ still respects the given constraints:

$$\begin{aligned}
\sum_{i=1}^{m_u} \rho''_{u,i} &= \sum_{i=1}^{m_u} \rho'_{u,i} = \rho_u; \\
\sum_{i=1}^{m_u} \frac{\rho''_{u,i}}{s_{u,i}} &= \left(\sum_{\substack{i=1 \\ i \neq i_{\min},\alpha,i_{\max}}}^{m_u} \frac{\rho'_{u,i}}{s_{u,i}}\right) + \frac{\rho''_{u,i_{\min}}}{s_{u,i_{\min}}} + \frac{\rho''_{u,\alpha}}{s_{u,\alpha}} + \frac{\rho''_{u,i_{\max}}}{s_{u,i_{\max}}} \\
&= \left(\sum_{i=1}^{m_u} \frac{\rho'_{u,i}}{s_{u,i}}\right) + \frac{\epsilon_{\min} + \epsilon_{\max}}{s_{u,\alpha}} - \frac{\epsilon_{\min}}{s_{u,i_{\min}}} - \frac{\epsilon_{\max}}{s_{u,i_{\max}}} \\
&= \left(\sum_{i=1}^{m_u} \frac{\rho'_{u,i}}{s_{u,i}}\right) + \epsilon_{\min}\left(\frac{1+\lambda}{s_{u,\alpha}} - \frac{1}{s_{u,i_{\min}}} - \frac{\lambda}{s_{u,i_{\max}}}\right) \\
&= \left(\sum_{i=1}^{m_u} \frac{\rho'_{u,i}}{s_{u,i}}\right) + \\
&\quad \frac{\epsilon_{\min}}{s_{u,i_{\min}} s_{u,\alpha}}\left(s_{u,i_{\min}} + \frac{s_{u,i_{\max}}(s_{u,\alpha} - s_{u,i_{\min}})}{(s_{u,i_{\max}} - s_{u,\alpha})} - s_{u,\alpha} - \frac{s_{u,\alpha}(s_{u,\alpha} - s_{u,i_{\min}})}{(s_{u,i_{\max}} - s_{u,\alpha})}\right) \\
&= \sum_{i=1}^{m_u} \frac{\rho'_{u,i}}{s_{u,i}} \leq \frac{1}{w}.
\end{aligned}$$

And the power consumed by the new solution is not greater than the original optimal one:

$$\begin{aligned}
\sum_{i=1}^{m_u} \rho'_{u,i}\mathfrak{K}_{u,i} - \sum_{i=1}^{m_u} \rho''_{u,i}\mathfrak{K}_{u,i} &= \epsilon_{\min}\mathfrak{K}_{u,i_{\min}} + \epsilon_{\max}\mathfrak{K}_{u,i_{\max}} - (\epsilon_{\min} + \epsilon_{\max})\mathfrak{K}_{u,\alpha} \\
&= \epsilon_{\min}(\mathfrak{K}_{u,i_{\min}} - \mathfrak{K}_{u,\alpha}) + \lambda\epsilon_{\min}(\mathfrak{K}_{u,i_{\max}} - \mathfrak{K}_{u,\alpha}) \\
&= \epsilon_{\min}(s_{u,\alpha} - s_{u,i_{\min}})\left(\frac{\mathfrak{K}_{u,i_{\min}} - \mathfrak{K}_{u,\alpha}}{s_{u,\alpha} - s_{u,i_{\min}}} + \frac{s_{u,i_{\max}}}{s_{u,i_{\min}}}\frac{\mathfrak{K}_{u,i_{\max}} - \mathfrak{K}_{u,\alpha}}{s_{u,i_{\max}} - s_{u,\alpha}}\right) \\
&\geq \epsilon_{\min}(s_{u,\alpha} - s_{u,i_{\min}})\left(\frac{\mathfrak{K}_{u,i_{\max}} - \mathfrak{K}_{u,\alpha}}{s_{u,i_{\max}} - s_{u,\alpha}} - \frac{\mathfrak{K}_{u,\alpha} - \mathfrak{K}_{u,i_{\min}}}{s_{u,\alpha} - s_{u,i_{\min}}}\right) \\
&\geq 0 \quad \text{because of the convexity of } \mathfrak{K}.
\end{aligned}$$

At each iteration, we set the throughput of either $i_{\min}$ or $i_{\max}$ to its throughput in $\tilde{\mathcal{S}}$, so the number of virtual processors which have different throughputs in $\mathcal{S}''$ and $\tilde{\mathcal{S}}$ is strictly decreasing. At the end, either one of the two other cases is reached so $\tilde{\mathcal{S}}$ does not consume more power than $\mathcal{S}''$, or $\tilde{\mathcal{S}} = \mathcal{S}''$. Overall, our scheduling is optimal.

■

Remark. *One can note that this gives a different proof from the one in [83], while using a general power consumption function, and while giving the formula of the throughput of each processor.*

Lemma 5.2. *The optimal power consumption to achieve a throughput of ρ is:*

$$\mathfrak{P}_u(\rho) = \max_{0 \leq i < m_u} \left\{ (w\rho - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \right\}.$$

Proof. The power consumption of $\tilde{\mathcal{S}}$ is:

$$\begin{aligned}
\mathfrak{P}_u(\rho) &= \tilde{\rho}_{i,i_0} \mathfrak{K}_{u,i_0} + \tilde{\rho}_{i,i_0+1} \mathfrak{K}_{u,i_0+1} \\
&= \frac{s_{u,i_0}(s_{u,i_0+1} - w\rho)}{w(s_{u,i_0+1} - s_{u,i_0})} \mathfrak{K}_{u,i_0} + \frac{s_{u,i_0+1}(w\rho - s_{u,i_0})}{w(s_{u,i_0+1} - s_{u,i_0})} \mathfrak{K}_{u,i_0+1} \\
&= \rho \frac{s_{u,i_0+1}\mathfrak{K}_{u,i_0+1} - s_{u,i_0}\mathfrak{K}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} - \frac{s_{u,i_0} s_{u,i_0+1} (\mathfrak{K}_{u,i_0+1} - \mathfrak{K}_{u,i_0})}{w(s_{u,i_0+1} - s_{u,i_0})} \\
&= w\rho \frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} - \frac{s_{u,i_0}\mathfrak{P}_{u,i_0+1} - s_{u,i_0+1}\mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} \\
&= w\rho \frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} - s_{u,i_0} \frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \mathfrak{P}_{u,i_0} \frac{s_{u,i_0+1} - s_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} \\
&= (w\rho - s_{u,i_0}) \frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \mathfrak{P}_{u,i_0}
\end{aligned}$$

As $\mathfrak{P}$ is super-linear, we have, if $j < k$:

$$\frac{\mathfrak{P}_{u,k} - \mathfrak{P}_{u,j}}{s_{u,k} - s_{u,j}} \geq \frac{\mathfrak{P}_{u,j+1} - \mathfrak{P}_{u,j}}{s_{u,j+1} - s_{u,j}} \quad \Rightarrow \quad \mathfrak{P}_{u,k} \geq (s_{u,k} - s_{u,j}) \frac{\mathfrak{P}_{u,j+1} - \mathfrak{P}_{u,j}}{s_{u,j+1} - s_{u,j}} + \mathfrak{P}_{u,j}$$

and, if $j > k$:

$$\frac{\mathfrak{P}_{u,j} - \mathfrak{P}_{u,k}}{s_{u,j} - s_{u,k}} \leq \frac{\mathfrak{P}_{u,j+1} - \mathfrak{P}_{u,j}}{s_{u,j+1} - s_{u,j}} \quad \Rightarrow \quad \mathfrak{P}_{u,k} \geq \mathfrak{P}_{u,j} - (s_{u,j} - s_{u,k}) \frac{\mathfrak{P}_{u,j+1} - \mathfrak{P}_{u,j}}{s_{u,j+1} - s_{u,j}}$$

As $s_{u,i_0} \leq w\rho_u \leq s_{u,i_0+1}$ and $\mathfrak{P}$ is super-linear, we have, for all if $s_{u,i_0} > s_{u,i}$:

$$\begin{aligned}
(w\rho - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} &= (w\rho - s_{u,i_0}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \\
&\quad \left((s_{u,i_0} - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \right) \\
&\leq (w\rho - s_{u,i_0}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i_0} \\
&\leq (w\rho - s_{u,i_0}) \frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \mathfrak{P}_{u,i_0} = \mathfrak{P}_u(\rho)
\end{aligned}$$

And, if $s_{u,i_0+1} \leq s_{u,i}$, so we have:

$$\begin{aligned}
(w\rho - s_{u,i})\frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} &\leq (w\rho - s_{u,i_0+1})\frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \\
&\quad \left(\mathfrak{P}_{u,i} - (s_{u,i} - s_{u,i_0+1})\frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}}\right) \\
&\leq (w\rho - s_{u,i_0+1})\frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i_0+1} \\
&\leq (w\rho - s_{u,i_0+1})\frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \mathfrak{P}_{u,i_0+1} \\
&\quad \textit{(* because } (w\rho - s_{u,i_0+1}) < 0\textit{*)} \\
&\leq (w\rho - s_{u,i_0})\frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \\
&\quad \left(\mathfrak{P}_{u,i_0+1} - (s_{u,i_0+1} - s_{u,i_0})\frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}}\right) \\
&\leq (w\rho - s_{u,i_0})\frac{\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}} + \mathfrak{P}_{u,i_0} = \mathfrak{P}_u(\rho)
\end{aligned}$$

Then i_0 is the mode that maximizes the formula:

$$(w\rho - s_{u,i})\frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}$$

■

Maximizing the throughput

Maximizing the throughput is a problem very similar to the one studied in the previous section. The maximization of the throughput is bounded by two types of constraints:

- The first constraint limits the total power consumption of the processors to a given bound,
- The second constraint states that the processing capacity of $P_{u,i}$ can not be exceeded, and that the different modes are exclusive.

$$\begin{cases}
\text{MAXIMIZE } \rho_u = \displaystyle\sum_{i=1}^{m_u} \rho_{u,i} \text{ SUBJECT TO} \\
\displaystyle\sum_{i=1}^{m_u} \frac{w\rho_{u,i}}{s_{u,i}}\mathfrak{P}_{u,i} \left(= \sum_{i=1}^{m_u} \rho_{u,i}\mathfrak{K}_{u,i}\right) \leq \mathfrak{P} \\
\displaystyle\sum_{i=1}^{m_u} \frac{w\rho_{u,i}}{s_{u,i}} \leq 1
\end{cases} \tag{5.22}$$

This linear program can be solved over the rationals, and this time the distribution of the frequencies of the processor will depend upon the bound on power consumption.

Lemma 5.3. *If $\mathfrak{P} > 0$ and $\exists\ i_0,\ 0 \leq i_0 < m_u,\ \mathfrak{P}_{u,i_0} < \mathfrak{P} \leq \mathfrak{P}_{u,i_0+1}$, then an optimal solution to maximize the throughput of the processor is:*

$$\tilde{\rho}_{u,i_0} = \frac{s_{u,i_0}(\mathfrak{P}_{u,i_0+1} - \mathfrak{P})}{w(\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0})} \qquad \tilde{\rho}_{u,i_0+1} = \frac{s_{u,i_0+1}(\mathfrak{P} - \mathfrak{P}_{u,i_0})}{w(\mathfrak{P}_{u,i_0+1} - \mathfrak{P}_{u,i_0})} \qquad \tilde{\rho}_{u,i} = 0 \text{ if } i \notin \{i_0, i_0+1\}$$

Otherwise the processor can run at its maximum mode ($\tilde{\rho}_{u,m_u} = \frac{s_{u,m_u}}{w}$) while respecting the power consumption constraint.

Proof. We first show that $\tilde{\boldsymbol{\mathcal{S}}}$ **is feasible**:

$$\begin{aligned}\sum_{i=1}^{m_u} \tilde{\rho}_{u,i}\frac{\mathfrak{P}_{u,i}w}{s_{u,i}} &= \frac{\mathfrak{P}_{u,i_0}(\mathfrak{P}_{u,i_0+1}-\mathfrak{P}) + \mathfrak{P}_{u,i_0+1}(\mathfrak{P}-\mathfrak{P}_{u,i_0})}{\mathfrak{P}_{u,i_0+1}-\mathfrak{P}_{u,i_0}} = \mathfrak{P};\\ \sum_{i=1}^{m_u} \frac{\tilde{\rho}_{u,i}w}{s_{u,i}} &= \frac{(\mathfrak{P}_{u,i_0+1}-\mathfrak{P})}{(\mathfrak{P}_{u,i_0+1}-\mathfrak{P}_{u,i_0})} + \frac{(\mathfrak{P}-\mathfrak{P}_{u,i_0})}{(\mathfrak{P}_{u,i_0+1}-\mathfrak{P}_{u,i_0})} = 1.\end{aligned}$$

Let $\mathcal{S}'$ be an optimal solution, $\mathcal{S}' = \{\rho'_{u,1}, \cdots, \rho'_{u,m_u}\}$. As $\mathcal{S}'$ is a solution of the linear program, it respects all the constraints. So:

$$\sum_{i=1}^{m_u} \rho'_{u,i}\mathfrak{K}_{u,i} \leq \mathfrak{P} \quad \text{and} \quad \sum_{i=1}^{m_u} \frac{\rho'_{u,i}w}{s_{u,i}} \leq 1.$$

Let $i_{\min}$ be the slowest mode used by $\mathcal{S}'$, and $i_{\max}$ the fastest. Then we can distinguish two cases:

- **If $\boldsymbol{i_{\min} > i_0}$ or $\boldsymbol{i_{\min} = i_0}$ and $\boldsymbol{\rho'_{u,i_0} = \tilde{\rho}_{u,i_0} - \epsilon_0 (\epsilon_0 > 0)}$:** (in both cases, $\rho'_{u,i_0} = \tilde{\rho}_{u,i_0} - \epsilon$) then we have:

$$\begin{aligned}\left(\sum_{i=i_0+1}^{m_u} \rho'_{u,i}\right)\mathfrak{K}_{u,i_0+1} &\leq \sum_{i=i_0+1}^{m_u} \rho'_{u,i}\mathfrak{K}_{u,i}\\ &\leq \mathfrak{P} - \rho'_{u,i_0}\mathfrak{K}_{u,i_0} = \mathfrak{P} - (\tilde{\rho}_{u,i_0}-\epsilon)\mathfrak{K}_{u,i_0}\\ &\leq (\mathfrak{P} - \tilde{\rho}_{u,i_0}\mathfrak{K}_{u,i_0}) + \epsilon\mathfrak{K}_{u,i_0} = \tilde{\rho}_{u,i_0+1}\mathfrak{K}_{u,i_0+1} + \epsilon\mathfrak{K}_{u,i_0}\\ &\leq \mathfrak{K}_{u,i_0+1}\left(\tilde{\rho}_{u,i_0+1} + \epsilon\frac{\mathfrak{K}_{u,i_0}}{\mathfrak{K}_{u,i_0+1}}\right)\\ \Rightarrow \sum_{i=i_0}^{m_u} \rho'_{u,i} &\leq \rho'_{u,i_0} + \sum_{i=i_0+1}^{m_u} \rho'_{u,i}\\ &\leq (\tilde{\rho}_{u,i_0}-\epsilon) + \left(\tilde{\rho}_{u,i_0+1} + \epsilon\frac{\mathfrak{K}_{u,i_0}}{\mathfrak{K}_{u,i_0+1}}\right)\\ &\leq \sum_{i=i_0}^{m_u} \tilde{\rho}_{u,i} - \epsilon\left(1 - \frac{\mathfrak{K}_{u,i_0}}{\mathfrak{K}_{u,i_0+1}}\right) \leq \sum_{i=i_0}^{m_u} \tilde{\rho}_{u,i}\end{aligned}$$

  And so our solution does not have a smaller throughput, and is thus also optimal.

- **If $\boldsymbol{i_{\max} < i_0 + 1}$ or $\boldsymbol{i_{\max} = i_0 + 1}$ and $\boldsymbol{\rho'_{u,i_0+1} = \tilde{\rho}_{u,i_0+1} - \epsilon_1 (\epsilon_1 > 0)}$:** (in both cases, $\rho'_{u,i_0+1} = \tilde{\rho}_{u,i_0+1} - \epsilon$) then, we have:

$$\begin{aligned}\frac{w\sum_{i=1}^{i_0}\rho'_{u,i}}{s_{u,i_0}} &\leq \sum_{i=1}^{i_0}\frac{\rho'_{u,i}w}{s_{u,i}} = \sum_{i=1}^{i_{\max}}\frac{\rho'_{u,i}w}{s_{u,i}} - \frac{\rho'_{u,i_0+1}w}{s_{u,i_0+1}}\\ &\leq 1 - \frac{\rho'_{u,i_0+1}w}{s_{u,i_0+1}} = \left(1 - \frac{\tilde{\rho}_{u,i_0+1}w}{s_{u,i_0+1}}\right) + \frac{\epsilon w}{s_{u,i_0+1}}\\ &\leq \frac{w\tilde{\rho}_{u,i_0}}{s_{u,i_0}} + \frac{\epsilon w}{s_{u,i_0+1}}\end{aligned}$$

So the throughput of $\mathcal{S}'$ is:

$$\begin{aligned}\sum_{i=1}^{i_{\max}} \rho'_{u,i} &\leq \sum_{i=1}^{i_0} \rho'_{u,i} + \rho'_{u,i_0+1} \\ &\leq \left(\tilde{\rho}_{u,i_0} + \epsilon \frac{s_{u,i_0}}{s_{u,i_0+1}}\right) + (\tilde{\rho}_{u,i_0} - \epsilon) \\ &\leq \sum_{i=1}^{m_u} \tilde{\rho}_{u,i} - \epsilon\left(1 - \frac{s_{u,i_0}}{s_{u,i_0+1}}\right) \leq \sum_{i=1}^{m_u} \tilde{\rho}_{u,i}\end{aligned}$$

And so our solution does not have a smaller throughput, and is thus also optimal.

- **Otherwise** we use the same new scheduling $\mathcal{S}''$ than is the previous section:

$$\begin{cases} \rho''_{u,i_{\min}} &= \rho'_{u,i_{\min}} - \epsilon_{\min} \\ \rho''_{u,\alpha} &= \rho'_{u,\alpha} + \epsilon_{\min} + \epsilon_{\max} \\ \rho''_{u,i_{\max}} &= \rho'_{u,i_{\max}} - \epsilon_{\max} \\ \rho''_{u,i} &= \rho'_{u,i} \quad \text{otherwise} \end{cases}$$

with $\epsilon_{\min} = \min\left\{\rho'_{u,i_{\min}}; \frac{\rho'_{u,i_{\max}}}{\lambda}\right\}$, $\epsilon_{\max} = \epsilon_{\min}\lambda$, and $\lambda = \frac{s_{u,i_{\max}}(s_{u,\alpha} - s_{u,i_{\min}})}{s_{u,i_{\min}}(s_{u,i_{\max}} - s_{u,\alpha})}$.

From the previous section, we know that $\mathcal{S}''$ does not consume more power than $\mathcal{S}'$, and so still respects the given constraints. And the throughput achieved is the same than $\mathcal{S}'$. By iterating this construction, we can extract an optimal scheduling where $i_{\min} = \alpha$ (each iteration sets the throughput of either $i_{\min}$ or $i_{\max}$ to zero).

■

Lemma 5.4. *The maximum achievable throughput according to the power consumption limit is:*

$$\rho_u(\mathfrak{P}) = \min\left\{\frac{s_{u,m_u}}{w}; \max_{1\leq i\leq m_u}\left\{\frac{\mathfrak{P}(s_{u,i+1} - s_{u,i}) + s_{u,i}\mathfrak{P}_{u,i+1} - s_{u,i+1}\mathfrak{P}_{u,i}}{w(\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i})}\right\}\right\}.$$

Proof. This proof is very similar to the one of lemma 5.2. ■

5.5.3 At the system level

In this section, we take the whole system into account, in order to either maximize the throughput or minimize the power consumption of the platform. But thanks to the previous section, we will be able to simplify the problem.

Minimization of the power consumption

If we started working on this problem from scratch, we would have written the following linear program (Equation (5.23)). The linear program is defined by five types of constraints:

- The first set of constraints links the throughput of a processor and the throughput of its virtual processors
- The second constraint states that the system has to ensure a given throughput,

- The third set of constraints states that the processing capacity of a processor P_u should not be exceeded, and that modes are exclusive.
- The fourth set of constraints states that the bandwidth of the link from P_{master} to P_u is not exceeded.
- The last constraint states that the total outgoing capacity of the master is not exceeded.

$$\begin{cases} \text{MINIMIZE } \mathfrak{P} = \displaystyle\sum_{u=1}^{p} \sum_{i=1}^{m_u} \rho_{u,i} \mathfrak{K}_{u,i} \text{ SUBJECT TO} \\ \forall u, \ \displaystyle\sum_{i=1}^{m_u} \rho_{u,i} = \rho_u \\ \displaystyle\sum_{u=1}^{p} \rho_u = \rho \\ \forall u, \ \displaystyle\sum_{i=1}^{m_u} \frac{\rho_{u,i}}{s_{u,u,i}} w \ \leq \ 1 \\ \forall u, \ \dfrac{\rho_u}{b_u} \delta \ \leq \ 1 \\ \displaystyle\sum_{u=1}^{p} \frac{\rho_u}{\text{BW}} \delta \ \leq \ 1 \end{cases} \tag{5.23}$$

This linear program is composed of a large number of equations ($3p+3$), and lots of variables ($\sum_{i=1}^{p} m_u \leq p \times \max_u\{m_u\}$). But thanks to the previous section, we no longer need to specify the throughput of each frequency of each processor. We only have to fix a throughput for each processor, and according to the previous section, we know how to use the different frequencies in order to achieve that throughput while minimizing the power constraint. Furthermore, the bounded multi-port constraint is not needed anymore, because either the outgoing capacity of the master is able to ensure the given throughput ($\text{BW} \geq \rho$), or the system has no solution. Overall, we can reduce the previous linear program to Equation (5.24):

$$\begin{cases} \text{MINIMIZE } \mathfrak{P} = \displaystyle\sum_{u=1}^{p} \mathfrak{P}_u \text{ SUBJECT TO} \\ \displaystyle\sum_{u=1}^{p} \rho_u = \rho \\ \forall u, \ \rho_u \ \leq \ \min\left\{ \dfrac{s_{u,m_u}}{w}; \dfrac{b_u}{\delta} \right\} \\ \forall \ u, \ \forall \ 1 \leq i \leq m_u, \ \mathfrak{P}_u \geq (w\rho_u - s_{u,i}) \dfrac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \end{cases} \tag{5.24}$$

Of course, the complexity of the problem has not totally disappeared. For each $\mathfrak{P}_u$ used in our objective function, we added m_u equations. But now we can reformulate the problem in such a way that it can easily be solved. The constraints of Linear program (5.24) state that we have to achieve a given throughput ρ, while the throughput of each processor is limited by its maximal speed and its bandwidth. The last constraint is just related with the power consumption, which has to be minimized. So this problem can be optimally solved using a greedy strategy: we sort all the processors in an increasing order according to their power consumption efficiency. Of

course, this power consumption depends on the different modes of the processors, and the same processor will appear a number of times equal to its number of modes. Formally, we sort in non decreasing order $\left\{\frac{\mathfrak{P}_{u,i+1}-\mathfrak{P}_{u,i}}{s_{u,i+1}-s_{u,i}}\right\}_{1\leq i\leq m_u, 1\leq u\leq p}$. The last step will be to select the cheapest virtual processors so the system can perform the given throughput, given that each processor's throughput will be limited by its maximal frequency and the bandwidth of the link between itself and the master. Altogether, we obtain the greedy Algorithm 11.

Algorithm 11: Greedy algorithm to minimize the power consumption given a throughput bound

Data: throughput ρ that has to be achieved
for $u = 1$ **to** p **do**
  $\mathcal{T}[u] \leftarrow 0$; /* *Throughput of processors* P_u */
$\Phi \leftarrow 0$; /* *Total throughput of the system* */
$\mathcal{L} \leftarrow$ sorted list of the P_{u_k,i_k} such that $\forall\, j,\ \frac{\mathfrak{P}_{u_j,1+i_j}-\mathfrak{P}_{u_j,i_j}}{s_{u_j,1+i_j}-s_{u_j,i_j}} \leq \frac{\mathfrak{P}_{u_{j+1},1+i_{j+1}}-\mathfrak{P}_{u_{j+1},i_{j+1}}}{s_{u_{j+1},1+i_{j+1}}-s_{u_{j+1},i_{j+1}}}$;
while $\Phi < \rho$ **do**
  $P_{u_k,i_k} \leftarrow \text{next}(\mathcal{L})$; /* *Selection of the next cheapest mode* */
  $\rho' \leftarrow \mathcal{T}[u_k]$; /* *previous throughput of* P_{u_k} *(at mode* $i_k - 1$*)* */
  $\mathcal{T}[u_k] \leftarrow \min\left\{\frac{s_{u_k,i_k}}{w}; \frac{b_{u_k}}{\delta}; \rho' + (\rho - \Phi)\right\}$; /* *new throughput of* P_{u_k} *(at mode* i_k*)* */
  if $\mathcal{T}[u_k] = \frac{b_{u_k}}{\delta}$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \backslash \{P_{u_k,j}\}$;/* *we do not need to look at faster modes of* P_{u_k} */
  $\Phi \leftarrow \Phi + \mathcal{T}[u_k] - \rho'$;

One can detail more precisely the line that gives the new throughput of P_{u_k} at mode i_k. One can easily understand that the throughput is bounded by the maximum throughput at this speed and by the maximum communication throughput. The last term states that the throughput does not have to be greater than the previous throughput (ρ') plus the remaining throughput that has to be achieved ($\rho - \Phi$).

One can notice that, if the last selected mode is $P_{u_{k_0},i_{k_0}}$, this greedy algorithm will

1. fully use each processor having at least one mode consuming strictly less than $P_{u_{k_0},i_{k_0}}$, and this either at the throughput of the bandwidth if reached (this throughput is achieved according to section 5.5.2), or at the largest single fastest mode that consumes strictly less than $P_{u_{k_0},i_{k_0}}$ or at the same mode than $P_{u_{k_0},i_{k_0}}$;
2. either not use at all or fully use at its first non-trivial mode any processor whose first non-trivial mode consumes exactly the same than $P_{u_{k_0},i_{k_0}}$;
3. not use at all any processor whose first non-trivial mode consumes strictly more than the mode $P_{u_{k_0},i_{k_0}}$;
4. use $P_{u_{k_0},i_{k_0}}$ at the minimum throughput so the system achieves a throughput of ρ, according to section 5.5.2.

Theorem 5.5. *Algorithm 11 optimally solves Linear program* (5.23).

Proof. Let $\tilde{\mathcal{S}} = \{\tilde{\rho}_u\}$ be the throughput of each processor given by Algorithm 11, and $\mathcal{S} = \{\rho_u\}$ be an optimal solution of the problem, different from $\tilde{\mathcal{S}}$. We know that there exists at least

one processor whose throughput in $\mathcal{S}$ is strictly lower that the one in $\tilde{\mathcal{S}}$, otherwise the power consumed by $\mathcal{S}$ would be greater than the one of $\tilde{\mathcal{S}}$. Let P_m be one of these processors. Of course, the remaining work of P_m in $\tilde{\mathcal{S}}$ has to be performed by (at least) one other processor, whose throughput is strictly greater in $\mathcal{S}$ than in $\tilde{\mathcal{S}}$ (otherwise, $\mathcal{S}$ could not achieve a total throughput of ρ). Let P_M be one of these processors.

The idea is then to transfer a portion of work from P_M to P_m. This amount of work ϵ equals to the minimum of the additional throughput needed by P_m to achieve a throughput $\tilde{\rho}_m$, and of the excess of throughput of P_M when compared to $\tilde{\mathcal{S}}$:

$$\epsilon = \min\{\tilde{\rho}_m - \rho_m; \rho_M - \tilde{\rho}_M\}.$$

What do we know about P_M in $\tilde{\mathcal{S}}$? We know for sure that Algorithm 11 required from it a throughput $\tilde{\rho}_M$ (which may be equal to 0). That means, according to the selection process of Algorithm 11, that: 1) either P_M is saturated by its bandwidth, but in that case, $\tilde{\rho}_M \geq \rho_M$, which contradicts the definition of P_M, or 2) P_M is saturated at a given mode $P_{M,i}$, and the next mode $P_{M,i+1}$ has a power consumption greater than, or equal to, any other selected processor, P_m included, or 3) P_M is not saturated, but in that case it is the last selected mode by Algorithm 11 and so has a power consumption greater than, or equal to, any other selected processor, P_m included. Overall, the power consumption of P_M is greater than, or equal to, the one of P_m.

Let $\mathcal{S}'$ be the scheduling where:

$$\begin{cases} \rho'_m &= \rho_m + \epsilon; \\ \rho'_M &= \rho_M - \epsilon; \\ \rho'_{i,j} &= \rho_{i,j} \text{ otherwise.} \end{cases}$$

Then, if we compare the power consumed by $\mathcal{S}'$ and $\mathcal{S}$:

$$\sum_{u=1}^{p} \mathfrak{P}'_u = \left(\sum_{\substack{u=1 \\ u \neq m,M}}^{p} \mathfrak{P}_u \right) + \mathfrak{P}'_m + \mathfrak{P}'_M.$$

$$\begin{aligned} \mathfrak{P}'_m &= \max_i \left\{ (w\rho'_m - s_{m,i}) \frac{\mathfrak{P}_{m,i+1} - \mathfrak{P}_{m,i}}{s_{m,i+1} - s_{m,i}} + \mathfrak{P}_{m,i} \right\} \\ &= \rho'_m \left(w \frac{\mathfrak{P}_{m,i_m+1} - \mathfrak{P}_{m,i_m}}{s_{m,i_m+1} - s_{m,i_m}} \right) + \left(\mathfrak{P}_{m,i_m} - s_{m,i_m} \frac{\mathfrak{P}_{m,i_m+1} - \mathfrak{P}_{m,i_m}}{s_{m,i_m+1} - s_{m,i_m}} \right) \\ &= (\rho_m + \epsilon)\lambda_{m_1} + \lambda_{m_2}, \\ \text{and } \mathfrak{P}'_M &= \max_i \left\{ (w\rho'_M - s_{M,i}) \frac{\mathfrak{P}_{M,i+1} - \mathfrak{P}_{M,i}}{s_{M,i+1} - s_{M,i}} + \mathfrak{P}_{M,i} \right\} \\ &= \rho'_M \left(w \frac{\mathfrak{P}_{M,i_M+1} - \mathfrak{P}_{M,i_M}}{s_{M,i_M+1} - s_{M,i_M}} \right) + \left(\mathfrak{P}_{M,i_M} - s_{M,i_M} \frac{\mathfrak{P}_{M,i_M+1} - \mathfrak{P}_{M,i_M}}{s_{M,i_M+1} - s_{M,i_M}} \right) \\ &= (\rho_M - \epsilon)\lambda_{M_1} + \lambda_{M_2}. \end{aligned}$$

We know that:

$$\frac{\mathfrak{P}_{m,i_m+1} - \mathfrak{P}_{m,i_m}}{s_{m,i_m+1} - s_{m,i_m}} \leq \frac{\mathfrak{P}_{M,i_{M+1}} - \mathfrak{P}_{M,i_M}}{s_{M,i_{M+1}} - s_{M,i_M}} \qquad \textit{(Greedy selection)}$$

So:

$$\begin{aligned}
\mathfrak{P}_m{}' &= (\rho_m\lambda_{m_1} + \lambda_{m_2}) + \epsilon\lambda_{m_1} \\
&= \mathfrak{P}_m + \epsilon\lambda_{m_1} \\
\mathfrak{P}_M{}' &= (\rho_M\lambda_{M_1} + \lambda_{M_2}) - \epsilon\lambda_{M_1} \\
&= \mathfrak{P}_M - \epsilon\lambda_{M_1} \\
\epsilon(\lambda_{m_1} - \lambda_{M_1}) &\leq 0 \\
\sum_{u=1}^{p} \mathfrak{P}'_u &\leq \left(\sum_{\substack{u=1 \\ u\neq m,M}}^{p} \mathfrak{P}_u \right) + \mathfrak{P}_m + \mathfrak{P}_M = \sum_{u=1}^{p} \mathfrak{P}_u
\end{aligned}$$

We can iterate these steps as long as $\mathcal{S}$ is different of $\tilde{\mathcal{S}}$, hence proving the optimality of our scheduling. ■

Maximizing the throughput

Maximizing the throughput is a very similar problem, and a similar greedy algorithm finds an optimal solution.

Algorithm 12: Greedy algorithm to maximize the throughput given a power consumption upper bound

Data: Power consumption lower bound $\mathfrak{P}$
for $u = 1$ **to** p **do**
  $\mathcal{T}[u] \leftarrow 0$;
$\Psi \leftarrow 0$;
$\mathcal{L} \leftarrow$ sorted list of the P_{u_k,i_k} such that $\forall\ j,\ \frac{\mathfrak{P}_{u_j,1+i_j} - \mathfrak{P}_{u_j,i_j}}{s_{u_j,1+i_j} - s_{u_j,i_j}} \leq \frac{\mathfrak{P}_{u_{j+1},1+i_{j+1}} - \mathfrak{P}_{u_{j+1},i_{j+1}}}{s_{u_{j+1},1+i_{j+1}} - s_{u_{j+1},i_{j+1}}}$;;
while $\Psi < \mathfrak{P}$ **do**
  $P_{u_k,i_k} \leftarrow \text{next}(\mathcal{L})$;
  $\mathfrak{P}' \leftarrow \mathcal{T}[u_k]$;
  $\mathcal{T}[u_k] \leftarrow \min\left\{\mathfrak{P}_{u_k,i_k}; \left(w\frac{b_{u_k}}{\delta} - s_{u_k,i_k}\right)\frac{\mathfrak{P}_{u_k,i_k+1} - \mathfrak{P}_{u_k,i_k}}{s_{u_k,i_k+1} - s_{u_k,i_k}} + \mathfrak{P}_{u_k,i_k}; \mathfrak{P}' + (\mathfrak{P} - \Psi)\right\}$;
  if $\mathcal{T}[u_k] = \left(w\frac{b_{u_k}}{\delta} - s_{u_k,i_k}\right)\frac{\mathfrak{P}_{u_k,i_k+1} - \mathfrak{P}_{u_k,i_k}}{s_{u_k,i_k+1} - s_{u_k,i_k}} + \mathfrak{P}_{u_k,i_k}$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \backslash \{P_{u_k,j}\}$;
  $\Psi \leftarrow \Psi + \mathcal{T}[u_k] - \mathfrak{P}'$;

On heterogeneous platforms, Algorithm 12 is optimal for maximizing the throughput given a bounded energy. The only difference between Algorithm 12 and Algorithm 11 is the objective function that is considered during the selection process. The proof of its optimality is then very similar to the one of Algorithm 11.

5.5.4 More realistic consumption models

When we move to more realistic platforms, the problem gets much more complicated.

Models

There exist many ways to improve the previous model in order to get a more realistic one. This includes the model where processors have a threshold mode (they cannot run slower than this mode), or the model with consumption overhead, and any combination of the previous models. We can have different problems when dealing with consumption overhead. First of all, we have to specify when the consumption overhead is paid, as one can have an overhead only when turning on the worker, when turning it off, or for each transition of mode; a processor turned on can consume even when idle. Of course, all combinations of the previous options are not realistic. Thus the case where the cost of tuning on and off a processor is null but it consumes even when it is idle is absurd: one just needs to turn it off when idle to have an apparent cost of zero. That is why we may need to add the constraint that processors can only be turned on once and never be turned off again. Many more problems come from timing overhead, or memory constraints. To understand the last point, one can consider multi-core processors. If at least one core is turned on, then one other core can be turned off and still some data can be sent to its memory. This way, the core will have data to process as soon as it is turned on. Of course, one has to declare whether the chosen model takes memory constraints into account.

Under these more realistic models, the power consumption depends now on the length of the interval during which the processor is turned on (we pay the overhead only once during this interval). We have to introduce some new notations to express the power consumption as a function of the length of the interval t we look at, and of the mode:

$$\mathfrak{P}_{u,i}(t) = \mathfrak{P}_{u,i}^{(1)} \cdot t + \mathfrak{P}_{u,i}^{(2)},$$

where $\mathfrak{P}_{u,i}^{(2)}$ is the energy overhead.

We will consider in the following sections that we have no memory constraints. Thus, we can adapt to the case with power overhead the closed formula given in Section 5.5.2 in order to determine the power consumption of processor P_u when running at throughput ρ_u during t time-units.

If there is no power overhead when switching between modes, then the formula is simply the same as previously:

$$\mathfrak{P}_u(t, \rho_u) = \max_{0 \leq i < m_u} \left\{ (w\rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1}(t) - \mathfrak{P}_{u,i}(t)}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}(t) \right\}$$

In that case, the throughput ρ_u, supposed to be between the two modes P_{u,i_0} and P_{u,i_0+1}, is obtained by running the mode P_{u,i_0} during $\frac{t(s_{u,i_0+1} - \rho_u w)}{s_{u,i_0+1} - s_{u,i_0}}$ time-units, and the mode P_{u,i_0+1} during $\frac{t(\rho_u w - s_{u,i_0})}{s_{u,i_0+1} - s_{u,i_0}}$ time-units (these values are obtained from the fraction of time the mode are used per time-unit in Section 5.5.2).

Otherwise, if we have power overhead when switching between modes, we have to compare the power consumption when switching between the modes with the power consumption of the upper mode $\mathfrak{P}_{u,i_0+1}(t)$.

Multi-core problem

In this part, we look at the problem of scheduling on multi-cores without memory constraints, where there is a power consumption overhead only when turning on the processor. Then, we can prove a property about one optimal schedule.

Lemma 5.5. *There exists an optimal schedule in which all processors, except possibly one, are used at a maximum throughput, i.e., either the throughput dictated by their bandwidth or the throughput of one of their modes.*

Proof. Let $\mathcal{S}$ be an optimal schedule without that property. We will study $\mathcal{S}$ during an interval of arbitrary length, say t time-units. As we have no control on the behavior of $\mathcal{S}$, every processor can be turned on and off as many times as possible. Let $\Delta_u(t)$ be the communication volume received by P_u during the t time-units, and $\Omega_u(t)$ the computational volume performed during this interval. One can note that both volumes are not necessary equal, as we chose an arbitrary time interval. We will now compare $\mathcal{S}$ and $\mathcal{S}'$, with $\mathcal{S}'$ being the schedule identical to $\mathcal{S}$ outside of the considered interval and which, during that interval, sends tasks to processors at rate $\frac{\Delta_u(t)}{t}$ and where processors compute with a throughput of $\frac{\Omega_u(t)}{t}$. We only need to focus to the most constrained problem, i.e., when the total communication throughput is lower than the total computational throughput. We suppose that the memory contains, at time $t = 0$, $\mathcal{M}_0$ tasks.

Communications under $\mathcal{S}'$ are feasible: Under $\mathcal{S}$, each processor received a volume of tasks equals to $\Delta_u(t)$ during t time-units, so its bandwidth throughput were greater than or equal to $\frac{\Delta_u(t)}{t}$, which means that $\mathcal{S}'$ also respects the bandwidth constraints. For the master's point of view, the total volume of communication during the t time-units under $\mathcal{S}$ is $\sum_{u=1}^{p} \Delta_u(t)$, so we had: $\sum_{u=1}^{p} \Delta_u(t) \leq t \cdot \text{BW}$. Consequently, $\sum_{u=1}^{p} \frac{\Delta_u(t)}{t}$ and $\mathcal{S}'$ respects the bounded capacity of the master.

Computations under $\mathcal{S}'$ are feasible: To perform the computation, we will use the technique described in the previous section. We have $\Omega_u(t) \leq \mathcal{M}_0 + \Delta_u(t)$. If

$$\frac{s_{u,i_0}}{w} \leq \frac{\Omega_u(t)}{t} \leq \frac{s_{u,i_0+1}}{w} \quad (i_0 \text{ might be equal to zero}),$$

then we use the mode P_{u,i_0} during

$$t_1 = \frac{t\left(s_{u,i_0+1} - \frac{\Omega_u(t)}{t}w\right)}{s_{u,i_0+1} - s_{u,i_0}}$$

time-units, and the mode P_{u,i_0+1} during

$$t_2 = \frac{t\left(\frac{\Omega_u(t)}{t}w - s_{u,i_0}\right)}{s_{u,i_0+1} - s_{u,i_0}}$$

time-units, and this solution is feasible. For the first part of the computation, the processor is run at its slower mode in order to minimize the power consumption. After t_1 time-units, the processor has stored a fraction of tasks equal to:

$$\mathcal{M}_0 + \left(\frac{\Delta_u(t)}{t} - \frac{s_{u,i_0}}{w}\right) t_1$$

Then, if we look at the memory $\mathcal{M}$ of the processor during the computation under the mode P_{u,i_0+1} after t' time-units ($t' \leq t_2$), we have:

$$\begin{aligned}
\mathcal{M} &= \mathcal{M}_0 + \left(\frac{\Delta_u(t)}{t} - \frac{s_{u,i_0}}{w}\right) t_1 - \left(\frac{s_{u,i_0+1}}{w} - \frac{\Delta_u(t)}{t}\right) t' \\
&\geq \mathcal{M}_0 + \left(\frac{\Delta_u(t)}{t} - \frac{s_{u,i_0}}{w}\right) t_1 - \left(\frac{s_{u,i_0+1}}{w} - \frac{\Delta_u(t)}{t}\right) t_2 \\
&= \mathcal{M}_0 + \frac{\Delta_u(t)}{t}(t_1+t_2) - \left(\frac{t_1 s_{u,i_0}}{w} + \frac{t_2 s_{u,i_0+1}}{w}\right) \\
&= \mathcal{M}_0 + \Delta_u(t)\left(\frac{s_{u,i_0+1} - \frac{\Omega_u(t)}{t}w}{s_{u,i_0+1} - s_{u,i_0}} + \frac{\frac{\Omega_u(t)}{t}w - s_{u,i_0}}{s_{u,i_0+1} - s_{u,i_0}}\right) \\
&\quad -t\left(\frac{s_{u,i_0}\left(s_{u,i_0+1} - \frac{\Omega_u(t)}{t}w\right)}{w(s_{u,i_0+1} - s_{u,i_0})} + \frac{s_{u,i_0+1}\left(\frac{\Omega_u(t)}{t}w - s_{u,i_0}\right)}{w(s_{u,i_0+1} - s_{u,i_0})}\right) \\
&= \mathcal{M}_0 + \Delta_u(t) - t\frac{\Omega_u(t)}{t} = \Omega_u(t) - \Omega_u(t) = 0
\end{aligned}$$

So there is always tasks in the memory of the processor, so $\mathcal{S}'$ is feasible.

$\mathcal{S}'$ does not consume more power than $\mathcal{S}$: We only pay a power overhead each time a processor is turned on, and $\mathcal{S}'$ turned on only once each processor used by $\mathcal{S}$. Furthermore, the throughput of each processor is smaller under $\mathcal{S}'$ than under $\mathcal{S}$, because the processors use under $\mathcal{S}'$ the whole t time-units to perform the same amount of work than under $\mathcal{S}$. Overall, the power consumption of $\mathcal{S}'$ is no greater than the one of $\mathcal{S}$.

Now, let us have a look at the throughput of each worker under $\mathcal{S}'$. If $\mathcal{S}'$ does not have the property of the lemma, then there are at least two processors P_m and P_M that are not at their maximum throughput. As $\mathcal{S}'$ uses the technique of section 5.5.2, we know that these throughput are achieved using only two modes P_{M,i_M} and P_{M,i_M+1} for P_M (P_{M,i_M} may have a throughput of zero), and P_{m,i_m}, P_{m,i_m+1} for P_m. Let us suppose that P_M consumes more power at its throughput than P_m at its own. This means that:

$$\frac{\mathfrak{P}_{m,i_m+1}(t) - \mathfrak{P}_{m,i_m}(t)}{s_{m,i_m+1} - s_{m,i_m}} \leq \frac{\mathfrak{P}_{M,i_M+1}(t) - \mathfrak{P}_{M,i_M}(t)}{s_{M,i_M+1} - s_{M,i_M}}.$$

We now construct a new schedule $\mathcal{S}''$ from $\mathcal{S}'$, with:

$$\mathcal{S}'' : \begin{cases}
\rho''_m &= \rho'_m + \epsilon \\
\rho''_M &= \rho'_M - \epsilon \\
\rho''_u &= \rho'_u \text{ otherwise} \\
\epsilon &= \min\left\{\frac{b_u}{\delta} - \rho'_m; \frac{s_{m,i_m+1}}{w} - \rho'_m; \rho'_M - \frac{s_{M,i_M}}{w}\right\}
\end{cases}$$

Then, if we compare the power consumed by $\mathcal{S}''$ and $\mathcal{S}'$:

$$\sum_{u=1}^{p} \mathfrak{P}''_u(t) = \left(\sum_{\substack{u=1 \\ u\neq m,M}}^{p} \mathfrak{P}'_u(t)\right) + \mathfrak{P}''_m(t) + \mathfrak{P}''_M(t).$$

$$\begin{aligned}
\mathfrak{P}''_m(t) &= \max_i \left\{ (w\rho''_m - s_{m,i}) \frac{\mathfrak{P}_{m,i+1}(t) - \mathfrak{P}_{m,i}(t)}{s_{m,i+1} - s_{m,i}} + \mathfrak{P}_{m,i}(t) \right\} \\
&= \rho''_m \left(w \frac{\mathfrak{P}_{m,i_m+1}(t) - \mathfrak{P}_{m,i_m}(t)}{s_{m,i_m+1} - s_{m,i_m}} \right) + \left(\mathfrak{P}_{m,i_m}(t) - s_{m,i_m} \frac{\mathfrak{P}_{m,i_m+1}(t) - \mathfrak{P}_{m,i_m}(t)}{s_{m,i_m+1} - s_{m,i_m}} \right) \\
&= (\rho'_m + \epsilon)\lambda_{m_1} + \lambda_{m_2}, \\
\text{and } \mathfrak{P}''_M(t) &= \max_i \left\{ (w\rho''_M - s_{M,i}) \frac{\mathfrak{P}_{M,i+1}(t) - \mathfrak{P}_{M,i}(t)}{s_{M,i+1} - s_{M,i}} + \mathfrak{P}_{M,i}(t) \right\} \\
&= \rho''_M \left(w \frac{\mathfrak{P}_{M,i_M+1}(t) - \mathfrak{P}_{M,i_M}(t)}{s_{M,i_M+1} - s_{M,i_M}} \right) + \left(\mathfrak{P}_{M,i_M}(t) - s_{M,i_M} \frac{\mathfrak{P}_{M,i_M+1}(t) - \mathfrak{P}_{M,i_M}(t)}{s_{M,i_M+1} - s_{M,i_M}} \right) \\
&= (\rho'_M - \epsilon)\lambda_{M_1} + \lambda_{M_2}.
\end{aligned}$$

As $\mathfrak{P}(t)$ is still super-linear (it is an affine function of $\mathfrak{P}$), we have:

$$\frac{\mathfrak{P}_{u,i_u+1}(t) - \mathfrak{P}_{u,i_u}(t)}{s_{u,i_u+1} - s_{u,i_u}} \leq \frac{\mathfrak{P}_{u,i_u+k+1}(t) - \mathfrak{P}_{u,i_u+k}(t)}{s_{u,i_u+k+1} - s_{u,i_u+k}}$$

So:

$$\begin{aligned}
\mathfrak{P}''_m(t) &= \left(\rho'_m \lambda_{m_1} + \lambda_{m_2}\right) + \epsilon\lambda_{m_1} \\
&= \mathfrak{P}'_m(t) + \epsilon\lambda_{m_1} \\
\mathfrak{P}''_M(t) &= \left(\rho'_M \lambda_{M_1} + \lambda_{M_2}\right) - \epsilon\lambda_{M_1} \\
&= \mathfrak{P}'_M(t) - \epsilon\lambda_{M_1} \\
\epsilon(\lambda_{m_1} - \lambda_{M_1}) &\leq 0 \\
\sum_{u=1}^{p} \mathfrak{P}''_u(t) &\leq \left(\sum_{\substack{u=1 \\ u \neq m,M}}^{p} \mathfrak{P}'_u(t) \right) + \mathfrak{P}'_m(t) + \mathfrak{P}'_M(t) = \sum_{u=1}^{p} \mathfrak{P}'_u(t).
\end{aligned}$$

Then $\mathcal{S}''$ achieves the same throughput than $\mathcal{S}'$, and does not consume more power than $\mathcal{S}'$. As the number of processor that are not at a maximum throughput is strictly smaller in $\mathcal{S}''$ than in $\mathcal{S}'$ as long as $\mathcal{S}'$ has two processors which are not at a maximum throughput, one can iterate this steps until only one processor is not saturated. ■

Remark (Memory constraints)**.** *In the presence of memory constraints, each processor has to switch to its upper mode when its memory is full. Our strategy is then to switch a minimum number of times. As we are in a steady-state approach, we can determine the cyclic behavior of a processor which has to achieve a throughput of ρ_u such that $\frac{s_{u,i_0}}{w} \leq \rho_u \leq \frac{s_{u,i_0+1}}{w}$. We suppose that the memory is empty at the beginning, and that the memory bound is $\mathcal{M}$. Then, the bandwidth throughput will be ρ_u, and the computation throughput will be either $\frac{s_{u,i_0}}{w}$ or $\frac{s_{u,i_0+1}}{w}$. Our goal is to run the slowest mode during the longest interval. After*

$$t_1 = \frac{\mathcal{M}}{\delta\left(\rho_u - \frac{s_{u,i_0}}{w}\right)}$$

time-units under the mode P_{u,i_0}, the memory will be full and we will have to switch to the mode P_{u,i_0+1} during

$$t_2 = \frac{\mathcal{M}}{\delta\left(\frac{s_{u,i_0+1}}{w} - \rho_u\right)}$$

time-units until the memory is empty, and to repeat this pattern.

We tried to adapt our greedy solution to take into account the power overhead (Algorithm 13); We saw previously that the performance of the algorithm depends on the time interval we consider. The greater it is, the less power overhead we have to pay. So our adapterd algorithm will sort all processors in an increasing order according to the ratio of their power consumption during d time-units at given mode over the speed of this mode, d being a parameter of the algorithm. We also compare this with the ratio of the power consumption of a processor when its throughput is limited by its bandwidth over this bandwidth.

Algorithm 13: Adapted Greedy algorithm

Input: d: time interval
Data: throughput ρ that has to be achieved
for $u = 1$ **to** p **do**
  $\mathcal{T}[u] \leftarrow 0$; /* *Throughput of processor* P_u */
$\Phi \leftarrow 0$; /* *Total throughput of the system* */
$\mathcal{L} \leftarrow$ sorted list of the (P_{u_k}, ρ_{u_k}) such that $\frac{\mathfrak{P}_{u_j,\rho_{u_j}}(d)}{\rho_{u_j}} \leq \frac{\mathfrak{P}_{u_{j+1},\rho_{u_{j+1}}}(d)}{\rho_{u_{j+1}}}$, with $\rho_{u_j} = \frac{s_{u_k,i_k}}{w}$ if $\frac{s_{u_k,i_k}}{w} < \frac{b_{u_k}}{\delta}$ and $\frac{b_{u_k}}{\delta}$ otherwise, ;
while $\Phi < \rho$ **do**
  $P_{u_k,i_k} \leftarrow \text{next}(\mathcal{L})$; /* *Selection of the next cheapest mode* */
  $\rho' \leftarrow \mathcal{T}[u_k]$; /* *previous throughput of* P_{u_k} */
  $\mathcal{T}[u_k] \leftarrow \min\{\rho_{u_k}; \rho' + (\rho - \Phi)\}$; /* *new throughput of* P_{u_k} */
  if $\mathcal{T}[u_k] = \frac{b_{u_k}}{\delta}$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \backslash \{P_{u_k,j}\}$;/* *we do not need to look at faster modes of* P_{u_k} */
  $\Phi \leftarrow \Phi + \mathcal{T}[u_k] - \rho'$;

Unfortunately, our algorithm is no longer optimal because of the last processor. During our selection process, we took into account the consumption of the processors during d time-units, so we know the consumption of all the first processors that we selected. But the last one may be limited by the remaining amount of work to perform, so it may not run during the whole interval or may not run at a constant throughput. And if the interval during which the processor has to run is smaller than that, then the ordering of the processors according to the ratio of their consumption over their throughput may change. For example, one can see that for achieving a very small throughput, we may prefer a processor with a small overhead and a large power consumption per time-units than the opposite.

Turned on only once

In this section, we will suppose that, once a processor is turned on, it will never be turned off, and thus will always have some non null power consumption. This model is more realistic, because a processor cannot be turned off and on at no cost. Under such constraints, the problem becomes NP-hard.

Definition 5.2 (MS-power). *Given two values $\mathcal{P}$ and ρ, a master-worker platform P composed of n workers, and a power consumption function $\mathfrak{P}$, is it possible to achieve a total throughput of at least ρ with a total power consumption at most equal to $\mathcal{P}$?*

First of all, our problem is harder than *MS-power*. If our problem of maximizing the throughput given an upper bound $\mathcal{P}$ on the power consumption (resp. minimizing the power consumption

given a lower bound ρ on the throughput) can be solved in polynomial time, then for all $\rho \leq \rho_{opt}$ (resp. $\mathcal{P} \geq \mathcal{P}_{\text{opt}}$) *MS-power* has a schedule, elsewhere it does not.

Theorem 5.6. *The problem MS-Power is NP-Hard on a platform where every processor can only be turned on once.*

Proof. We will make the reduction from 2-partition.

First of all, it is clear that the problem belongs to NP.

Let $I = \{a_1, ..., a_n\}$ be an instance of 2-Partition. We will note $2A = \sum_{i=1}^{n} a_i$. We remind the reader that the objective of 2-Partition is to find one subset of I whose sum is equal to A.

Let I' be an instance of *MS-power*. We build a platform composed of a master and n workers, each worker having a unique mode of speed a_i, a bandwidth of a_i, and a bound on the master's bandwidth equal to $2A$. Processor P_i as a power consumption of a_i per time-unit at its only mode. Finally, we let $\mathcal{P} = \rho = A$.

Then, if we find a scheduling whose power consumption is A and whose throughput is A, we have exhibited a subset of I whose sum is equal to A. On the other hand, if one can find a subset of I whose sum is equal to A, then it represents a feasible schedule. ■

One can note that lemma 5.5 also holds for this problem: at most one processor is not used at a maximum throughput. For this problem, we adapt our original greedy algorithm so that the last processor to be selected is used at the throughput determined by the greedy algorithm only if this throughput is greater than or equal to its first mode. Otherwise, if we want to minimize the power consumption given a lower bound on the throughput, we compare the power consumption of all processors when having an additional amount of work equal to the remaining throughput that has to be achieved, and we pick the best solution. On the contrary, if we want to maximize the throughput given an upper bound on the power consumption, then we discard from $\mathcal{L}$ this last processor and all processors that were not yet enrolled by the greedy algorithm, and then we continue the execution of the greedy algorithm. In other words, we are looking among the enrolled processors the processors (more than one may be chosen) that can most efficiently use the remaining energy budget. We compute the power consumption each processor would have if we request it to increase its throughput by the "missing" throughput (the difference between the required throughput and the throughput achieved so far).

5.5.5 Related Work

Several papers have been targeting the minimization of power consumption. Most of these works suppose that they can switch to arbitrary speed values. Here is a brief overview of those papers:

Unit time tasks. Bunder in [38] 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 chooses to have a continuous range of speeds for the processors. He extends his work from one processor to multi-processors, but unlike us, did not take any communication time into account. His approach corresponds to scheduling on multi-core processors. He also proves the NP-completeness of the problem of minimizing the makespan on multi-processors with jobs of different amount of work.

Authors in [4] concentrate on minimizing the total flow time of unit time jobs with release dates on one processor. After proving that no online algorithm can achieve a constant

competitive ratio if job have arbitrary sizes, they exhibit a constant competitive online algorithm and solve the offline problem in polynomial time. Contrarily to [38] where the authors gather the tasks into blocks and schedule them with increasing speed in order to minimize the makespan, here the authors prove that the speed of the blocks need to be decreasing in order to minimize both total flow time and the energy consumption.

Communication-aware. In [142], the authors are interested about scheduling task graphs with data dependences while minimizing the energy consumption of both the processors and the interprocessor communication devices. They demonstrate that in the context of multiprocessor systems, the interprocessor communications were an important source of consumption, and their algorithm reduces up to 80% the communications. However, as they focus on multiprocessor problems, they only consider the energy consumption of the communications, and suppose that the communication times are negligible compared to the computation times.

Deadlines. Most papers are trying to minimize the energy consumed by the platform given a set of deadlines for all tasks on the system. In [119], the authors focus on the problem where tasks arrive according to some release dates. They show that during each time interval composed of the release dates and the deadlines of the applications, the optimal voltage is constant, and they determine this voltage, as well as the minimum constant speed for each job.

[10] improves the best known competitive ratio to minimize the energy while respecting all deadlines.

[46] works on a overloaded platform (which means that no algorithm can finish all the jobs) and try to maximize the throughput. Their online algorithm is $O(1)$ competitive for both throughput maximization and energy minimization.

Task-related consumption. [5] addresses the problem of periodic independent real-time tasks on one processor, the period being a deadline to all tasks. The particularity of this work is that they suppose the energy consumption is related to the *task* that is executed on the processor. They exhibit a polynomial algorithm to find the optimal speed of each task, and proved that EDF can be used to obtain a feasible schedule with these optimal speed values.

Discrete voltage case. In [83], the authors represent the problem of scheduling tasks on a single processor with discrete voltage. They also look at the model where the energy consumption is related to the *task*, and describe how to split the voltage for each task. They extend their work in [110] to online problems.

In [147], the authors add the constraint that the voltage can only be changed at each cycle of every task, in order to limit the number of transitions and thus the energy overhead. They find that under such model, the minimal number of frequency's transitions in order to minimize the energy may be greater than two.

Slack sharing. In [148, 122], the authors investigate dynamic scheduling. They are dealing with the problem of scheduling DAG tasks before deadlines, using a semi-clairvoyant model. For each task, the only information available is their worst-case execution times. Their algorithm operates in two steps: first a greedy static algorithm schedules the tasks on the processors according to their worst-case execution times and the deadline, and reduces

the processors speed so each processor meets the deadline. Then, if a task ends sooner than according to the static algorithm, a dynamic slack sharing algorithm uses the extra-time to reduce the speed of computations for the following tasks. The authors investigate the problem with time overhead and voltage overhead when changing processor speeds, and adapt their algorithm accordingly. Unfortunately, they do not take into account any communication.

5.6 Conclusion

In the first part of this chapter, we have studied the problem of scheduling multiple applications, made of collections of independent and identical tasks, on a heterogeneous master-worker platform. These applications had different release dates. We aimed at minimizing the maximum stretch, or equivalently at minimizing the largest relative slowdown of each application due to their concurrent executions. We derived an optimal algorithm for the off-line setting (when all application sizes and release dates are known beforehand). We have adapted this algorithm to an online scenario, so that it can react when new applications are released.

We have evaluated in practice our new algorithms against classical greedy heuristics, and also against some involved static multi-applications strategies. This evaluation was made both through actual experiments on a real cluster, using MPI, and through extensive simulations, conducted with SimGrid. Both types of evaluation showed a great improvement when using our **CBS3M** strategy, which achieves an averaged worse max-stretch only 16% greater than the off-line optimal max-stretch To the best of our knowledge, this work is the first attempt to provide efficient scheduling techniques for multiple bag-of-tasks applications in an online scenario.

Future work includes extending the approach to other communication models (such as the one-port model) and to more general platforms (such as multi-level trees). It would also be very interesting to deal with more complex application types, such as pipelines or even general DAGs.

In the second part, we have studied the problem of scheduling a single application with power consumption constraints, on a heterogeneous master-worker platform. We proved some relationships between the throughput and the power consumption at the processor level, and we developed an optimal algorithm for the problem on the whole heterogeneous platform for the simple ideal power consumption model. When we changed our power consumption model to a more realistic one, problems became more complicated. We proved that some problems became NP-Hard, and we heuristically extended our algorithm to deal with these models.

We would like to point out that Section 5.5 is still an ongoing work. As future work, it would be interesting to develop our study towards more realistic models, and find some approximation algorithms. It would then be necessary to test them through simulations.

Chapter 6

Conclusion

Throughout this thesis, we tried to underline the challenges of scheduling on heterogeneous platforms with realistic communication models. Even very simple problems such as scheduling independent identical tasks on master-worker platforms proved to be hard in practice. However, the simplicity of the platform's shape under the master-worker model helped us to find greedy algorithms which achieve good performance.

The main contributions of this work can be divided in three parts.

Scheduling : we proved some new results for online and offline scheduling of independent identical tasks with release dates on heterogeneous platforms. For the online setting, we have provided a comprehensive set of lower bounds for the competitive ratio of any deterministic online scheduling algorithm, for each source of heterogeneity and for three objective functions. For the offline setting, we have derived several new results, as an optimal makespan-minimization algorithm for communication-homogeneous platform, and an NP-hardness proof of the scheduling problem on fully heterogeneous platform.

Matrix product : we studied the problem of performing a matrix product on heterogeneous master-worker platforms with the constraints of centralized data and limited memory. As the problem of minimizing the makespan proved to be difficult, we focused on the minimization of the total communication volume. We have derived a new, tighter, bound for the minimal volume of communications needed to perform the multiplication, and developed an efficient memory layout, i.e., an algorithm to share the memory available on the workers among the three matrices. We also extended this algorithm to provide an efficient solution for heterogeneous platforms, as demonstrated by our MPI experiments.

Steady-state : as the difficulty of scheduling was related to the metric used (the makespan), we focused at last on steady-state scheduling. In a first part, we have studied the problem of scheduling multiple applications with release dates on a heterogeneous master-worker platform. We aimed at minimizing the largest relative slowdown of each application due to their concurrent execution. We derived an optimal algorithm for the off-line setting, and we have adapted this algorithm to an online scenario, so that it can react when new applications are released. In a second part, we studied the problem of maximizing the throughput of one application given an upper bound on the power consumption, and the

problem of minimizing the power consumption given a lower bound on the throughput, in both cases on a platform composed of multi-modes processors. We developed an optimal algorithm for the problem under the simple ideal power consumption model. Under more realistic models, we have proved one NP-completeness result, and extended our algorithm.

Along these chapters, we have found efficient algorithms while minimizing the makespan. In fact, it may be a better approach to focus on another objective function even if the real objective is the minimization of the total completion time. For example, we focused in Chapter 4 on the minimization of the total communication volume, as we thought that it was the bottleneck of the application execution time. This objective function was easier to study, and the algorithms obtained had at the end a smaller completion time than others. Another example occurred in Chapter 5 when we focused on the minimization of the stretch of several applications. Using such a metric led to solutions whose platform utilizations were better, and so the total completion time was the best among all other algorithms.

We also assumed that we had the exact communication speeds and computation speeds of the workers. However, a realist hypothesis would be that we only have estimations, more or less accurate. This is especially true for the communication times, because one has to share the physical network links among several concurrent applications. For example, we often refereed to BOINC-like computations, because they are composed of multiple bags of independent identical tasks, and because participating processors are organized into a master-worker platform. However, BOINC does not know the communication bandwidth between the server and the workers. In such a context, one may need to change our platform model. From the computational point of view, one can also think of the scenario of platforms where the speed of the machines can vary through time, and these machines may even be turned off, because of physical problems, at any time during the scheduling. When one want to solve such problems, dynamic strategies and a more decentralized approach may be a solution.

Next, one can remark that all previous studies were possible thanks to the simplicity of the master-worker model. We often insist on the importance of taking into account the communication times, but we have neglected the contention problems between the network links. Most of the time, it was a reasonable assumption, because of the one-port model, but we could discuss it further when using the bounded multi-port model.

So an important question to answer is: “if I do not have a master-worker platform, i.e., a star network between the resources, what should I do?”. The question is to determine the best way to map a master-worker platform (i.e., a star network between the master and the workers) from any network connection graph. What do we do in the case where the machines are linked by a sparse network? Do we choose the master and the workers so as to enroll the largest number of workers on a star-network composed of these resources so that no contention will occur ? Or do we change our star-network to a multi-level tree ?

These are some important issues to investigate if one wants to squeeze the most out of the various resources at our disposal.

Appendix A

Proofs of online competitiveness

Communication-homogeneous platforms

In this section, we have different-speed processors, so we order them such that P_1 is the fastest processor (p_1 is the smallest computing time p_i), while P_m is the slowest processor.

Theorem A.1. *There is no scheduling algorithm for the problem*

$$Q_{>1}, MS \mid online,\ r_i,\ p_j,\ c_j = c \mid \max\ C_i$$

with a competitive ratio less than $\frac{5}{4}$.

Proof. Suppose the existence of an on-line algorithm $\mathcal{A}$ with a competitive ratio $\rho = \frac{5}{4} - \epsilon$, with $\epsilon > 0$. We will build a platform and study the behavior of $\mathcal{A}$ opposed to our adversary. The platform consists of two processors, where $p_1 = 3$, $p_2 = 7$, and $c = 1$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ sends the task i either to P_1, achieving a makespan at least equal to $c + p_1 = 4$, or on P_2, with a makespan at least equal to $c + p_2 = 8$. At time $t_1 = c$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and the adversary reacts consequently:

1. If $\mathcal{A}$ did not begin the sending of the task i, the adversary does not send other tasks. The best makespan is then $t_1 + c + p_1 = 5$. As the optimal makespan is 4, we have a competitive ratio of $\frac{5}{4} > \rho$. This refutes the assumption on ρ. Thus the algorithm $\mathcal{A}$ must have scheduled the task i at time c.

2. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible makespan is then equal to $c + p_2 = 8$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have another choice than to schedule the task i on P_1 in order to be able to respect its competitive ratio.

At time $t_1 = c$, the adversary sends another task, j. In this case, we look, at time $t_2 = 2c$, at the assignment $\mathcal{A}$ made for j:

1. If j is sent on P_2, the adversary does not send any more task. The best achievable makespan is then $\max\{c + p_1, 2c + p_2\} = \max\{1 + 3, 2 + 7\} = 9$, whereas the optimal is to send the two tasks to P_1 for a makespan of $\max\{c + 2p_1, 2c + p_1\} = 7$. The competitive ratio is then $\frac{9}{7} > \frac{5}{4} > \rho$.

2. If j is sent on P_1 the adversary sends a last task at time $t_2 = 2c$. Then the schedule has the choice to execute the last task either on P_1 for a makespan of $\max\{c+3p_1, 2c+2p_1, 3c+p_1\} = \max\{10, 8, 6\} = 10$, or on P_2 for a makespan of $\max\{c+2p_1, 2c+p_1, 3c+p_2\} = \max\{6, 5, 10\} = 10$. The best achievable makespan is then 10. However, scheduling the first task on P_2 and the two others on P_1 leads to a makespan of $\max\{c+p_2, 2c+2p_1, 3c+p_1\} = \max\{8, 8, 6\} = 8$. The competitive ratio is therefore at least equal to $\frac{10}{8} = \frac{5}{4} > \rho$.

3. If j is not sent yet, then the adversary sends a last task at time $t_2 = c_2$. $\mathcal{A}$ has the choice to execute j on P_1, and to achieve a makespan worse than the previous case, or on P_2. And it has then the choice to send k either to P_2 for a makespan of $\max\{c+p_1, t_2+c+p_2+\max\{c, p_2\}\} = \max\{4, 17\} = 17$, or to P_1 for a makespan of $\max\{c+2p_1, t_2+c+p_2, t_2+2c+p_1\} = max\{7, 10, 7\} = 10$. The best achievable makespan is then 10. The competitive ratio is therefore at least equal to $\frac{10}{8} = \frac{5}{4} > \rho$. Hence the desired contradiction.

■

Theorem A.2. *There is no scheduling algorithm for the problem*

$$Q_{>1}, MS \mid online,\ r_i,\ p_j,\ c_j = c \mid \max\ (C_i - r_i)$$

with a competitive ratio less than $\frac{5-\sqrt{7}}{2}$.

Proof. Suppose the existence of an on-line algorithm $\mathcal{A}$ with a competitive ratio $\rho = \frac{5-\sqrt{7}}{2} - \epsilon$, with $\epsilon > 0$. We will build a platform and study the behavior of $\mathcal{A}$ opposed to our adversary. The platform consists of two processors, where $p_1 = \frac{2+\sqrt{7}}{3}$, $p_2 = \frac{1+2\sqrt{7}}{3}$, and $c = 1$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ sends the task i either to P_1, achieving a max-flow at least equal to $c + p_1 = \frac{5+\sqrt{7}}{3}$, or on P_2, with a max-flow at least equal to $c + p_2 = \frac{4+2\sqrt{7}}{3}$. At time $\tau = \frac{4-\sqrt{7}}{3}$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and the adversary reacts consequently:

1. If $\mathcal{A}$ did not begin the sending of the task i, the adversary does not send other tasks. The best possible max-flow is then $\tau + c + p_1 = 3$. As the optimal max-flow is $\frac{5+\sqrt{7}}{3}$, we have a competitive ratio of $\frac{9}{5+\sqrt{7}} = \frac{5-\sqrt{7}}{2} > \rho$. This refutes the assumption on ρ. Thus the algorithm $\mathcal{A}$ must have scheduled the task i at time τ.

2. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible max-flow is then equal to $\frac{4+2\sqrt{7}}{3}$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have another choice than to schedule the task i on P_1 in order to be able to respect its competitive ratio.

At time $\tau = \frac{4-\sqrt{7}}{3}$, the adversary sends another task, j. The best schedule would have been to send the first task to P_2 and the second to P_1 achieving a max-flow of $\max \leq \{c+p_2, 2c+p_1-\tau\} = \max\left\{\frac{4+2\sqrt{7}}{3}, \frac{4+2\sqrt{7}}{3}\right\} = \frac{4+2\sqrt{7}}{3}$. We look at the assignment $\mathcal{A}$ made for j:

1. If j is sent on P_2, the best achievable max-flow is then $\max\{c + p_1, 2c + p_2 - \tau\} = \max\{\frac{5+\sqrt{7}}{3}, 1 + \sqrt{7}\} = 1 + \sqrt{7}$, whereas the optimal is $\frac{4+2\sqrt{7}}{3}$. The competitive ratio is then $\frac{5-\sqrt{7}}{2} > \rho$.

2. If j is sent on P_1, the best possible max-flow is then $\max\{c+p_1, \max\{c+2p_1, 2c+p_1\}-\tau\} = \max\{\frac{5+\sqrt{7}}{3}, 1+\sqrt{7}\} = 1+\sqrt{7}$. The competitive ratio is therefore once again equal to $\frac{5-\sqrt{7}}{2} > \rho$.

■

Theorem A.3. *There is no scheduling algorithm for the problem*

$$Q_{>1}, MS \mid online,\ r_i,\ p_j,\ c_j = c \mid \sum (C_i - r_i)$$

with a competitive ratio less than $\frac{2+4\sqrt{2}}{7}$.

Proof. Suppose the existence of an on-line algorithm $\mathcal{A}$ with a competitive ratio $\rho = \frac{2+4\sqrt{2}}{7} - \epsilon$, with $\epsilon > 0$. We will build a platform and study the behavior of $\mathcal{A}$ opposed to our adversary. The platform consists of two processors, where $p_1 = 2$, $p_2 = 4\sqrt{2} - 2$, and $c = 1$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ sends the task i either to P_1, achieving a sum-flow at least equal to $c + p_1 = 3$, or on P_2, with a sum-flow at least equal to $c+p_2 = 4\sqrt{2}-1$. At time $t_1 = c$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and the adversary reacts consequently:

1. If $\mathcal{A}$ did not begin the sending of the task i, the adversary does not send other tasks. The best sum-flow is then $t_1 + c + p_1 = 4$. As the optimal sum-flow is 3, we have a competitive ratio of $\frac{4}{3} > \rho$. This refutes the assumption on ρ. Thus the algorithm $\mathcal{A}$ must have scheduled the task i at time c.

2. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible sum-flow is then equal to $c + p_2 = 4\sqrt{2} - 1$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have another choice than to schedule the task i on P_1 in order to be able to respect its competitive ratio.

At time $t_1 = c$, the adversary sends another task, j. In this case, we look, at time $t_2 = 2c$, at the assignment $\mathcal{A}$ made for j:

1. If j is sent to P_2, the adversary does not send any more task. The best achievable sum-flow is then $(c+p_1) + ((2c+p_2) - t_1) = 2 + 4\sqrt{2}$, whereas the optimal is to send the two tasks to P_1 for a sum-flow of $(c+p_1) + (\max\{2c+p_1, c+2p_1\} - t_1) = 7$. The competitive ratio is then $\frac{2+4\sqrt{2}}{7} > \rho$.

2. If j is sent to P_1 the adversary sends a last task at time $t_2 = 2c$. Then the schedule has the choice to execute the last task either on P_1 for a sum-flow of $(c + p_1) + (\max\{c + 2p_1, 2c+p_1\} - t_1) + (\max\{3c+p_1, c+3p_1\} - t_2) = 12$, or on P_2 for a sum-flow of $(c+p_1) + (\max\{c+2p_1, 2c+p_1\} - t_1) + ((3c+p_2) - t_2) = 6 + 4\sqrt{2}$. The best achievable sum-flow is then $6 + 4\sqrt{2}$. However, scheduling the *second* task on P_2 and the two others on P_1 leads to a sum-flow of $(c+p_1) + ((2c+p_2) - t_1) + (\max\{3c+p_1, c+2p_1\} - t_2) = 5 + 4\sqrt{2}$. The competitive ratio is therefore at least equal to $\frac{6+4\sqrt{2}}{5+4\sqrt{2}} = \frac{2+4\sqrt{2}}{7} > \rho$.

3. If j is not send yet, then the adversary sends a last task k at time $t_2 = 2c$. Then the schedule has the choice to execute j either on P_1, and achieving a sum-flow worse than the previous case, or on P_2. Then, it can choose to execute the last task either on P_2 for a

sum-flow of $(c+p_1)+(t_2+c+p_2-t_1)+(t_2+c+p_2+\max\{c,p_2\}-t_2) = 12\sqrt{2}+2$, or on P_1 for a sum-flow of $(c+p_1)+(t_2+c+p_2-t_1)+(\max\{t_2+c+p_1, c+2p_1\}-t_2) = 7+4\sqrt{2}$. The best achievable sum-flow is then $7+4\sqrt{2}$ which is even worse than the previous case. Hence the desired contradiction.

■

Computation-homogeneous platforms

In this section, we have $p_j = p$ but processor links with different capacities. We order them, so that P_1 is the fastest communicating processor (c_1 is the smallest computing time c_i).

Theorem A.4. *There is no scheduling algorithm for the problem*

$$P_{>1}, MS \mid online,\ r_i,\ p_j = p,\ c_j \mid \max C_i$$

whose competitive ratio ρ is strictly lower than $\frac{6}{5}$.

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \frac{6}{5} - \epsilon$, with $\epsilon > 0$. We will build a platform and an adversary to derive a contradiction. The platform is made up with two processors P_1 and P_2 such that $p_1 = p_2 = p = \max\left\{5, \frac{24}{25\epsilon}\right\}$, $c_1 = 1$ and $c_2 = \frac{p}{2}$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i, either on P_1 with a makespan at least equal to $1+p$, or on P_2 with a makespan at least equal to $\frac{3p}{2}$.

At time-step $\frac{p}{2}$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible makespan is then $\frac{3p}{2}$. The optimal scheduling being of makespan $1+p$, we have a competitive ratio of
$$\rho \geq \frac{\frac{3p}{2}}{1+p} = \frac{3}{2} - \frac{3}{2(p+1)} > \frac{6}{5}$$
because $p \geq 5$ by assumption. This contradicts the hypothesis on ρ. Thus the algorithm $\mathcal{A}$ cannot schedule task i on P_2.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send other tasks. The best makespan that can be achieved is then equal to $1+\frac{p}{2}+p = 1+\frac{3p}{2}$, which is even worse than the previous case. Consequently, the algorithm $\mathcal{A}$ does not have any other choice than to schedule task i on P_1.

At time-step $\frac{p}{2}$, the adversary sends three tasks, j, k and l. No schedule which executes three of the four tasks on the same processor can have a makespan lower than $1+3p$ (minimum duration of a communication and execution without delay of the three tasks). We now consider the schedules which compute two tasks on each processor. Since i is computed on P_1, we have three cases to study, depending upon which other task (j, k, or l) is computed on P_1:

1. If j is computed on P_1, at best we have:
   (a) Task i is sent to P_1 during the interval $[0,1]$ and is computed during the interval $[1, 1+p]$.

(b) Task j is sent to P_1 during the interval $[\frac{p}{2}, 1+\frac{p}{2}]$ and is computed during the interval $[1+p, 1+2p]$.

(c) Task k is sent to P_2 during the interval $[1+\frac{p}{2}, 1+p]$ and is computed during the interval $[1+p, 1+2p]$.

(d) Task l is sent to P_2 during the interval $[1+p, 1+\frac{3p}{2}]$ and is computed during the interval $[1+2p, 1+3p]$.

The makespan of this schedule is then $1+3p$.

2. If k is computed on P_1:

(a) Task i is sent to P_1 during the interval $[0, 1]$ and is computed during the interval $[1, 1+p]$.

(b) Task j is sent to P_2 during the interval $[\frac{p}{2}, p]$ and is computed during the interval $[p, 2p]$.

(c) Task k is sent to P_1 during the interval $[p, 1+p]$ and is computed during the interval $[1+p, 1+2p]$.

(d) Task l is sent to P_2 during the interval $[1+p, 1+\frac{3p}{2}]$ and is computed during the interval $[2p, 3p]$.

The makespan of this scheduling is then $3p$.

3. If l is computed on P_1:

(a) Task i is sent to P_1 during the interval $[0, 1]$ and is computed during the interval $[1, 1+p]$.

(b) Task j is sent to P_2 during the interval $[\frac{p}{2}, p]$ and is computed during the interval $[p, 2p]$.

(c) Task k is sent to P_2 during the interval $[p, \frac{3p}{2}]$ and is computed during the interval $[2p, 3p]$.

(d) Task l is sent to P_1 during the interval $[\frac{3p}{2}, 1+\frac{3p}{2}]$ and is computed during the interval $[1+\frac{3p}{2}, 1+\frac{5p}{2}]$.

The makespan of this schedule is then $3p$.

Consequently, the last two schedules are equivalent and are better than the first. Altogether, the best achievable makespan is $3p$. But a better schedule is obtained when computing i on P_2, then j on P_1, then k on P_2, and finally l on P_1. The makespan of the latter schedule is equal to $1+\frac{5p}{2}$. The competitive ratio of algorithm $\mathcal{A}$ is necessarily larger than the ratio of the best reachable makespan (namely $3p$) and the optimal makespan, which is not larger than $1+\frac{5p}{2}$. Consequently:

$$\rho \geq \frac{3p}{1+\frac{5p}{2}} = \frac{6}{5} - \frac{12}{5(5p+2)} > \frac{6}{5} - \frac{12}{25p} \geq \frac{6}{5} - \frac{\epsilon}{2}$$

which contradicts the assumption $\rho = \frac{6}{5} - \epsilon$ with $\epsilon > 0$. ■

Theorem A.5. *There is no scheduling algorithm for the problem*

$$P_{>1}, MS \mid online\ r_i,\ p_j = p,\ c_j \mid \max\,(C_i - r_i)$$

whose competitive ratio ρ is strictly lower than $\frac{5}{4}$.

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \frac{5}{4} - \epsilon$, with $0 < \epsilon \leq \frac{1}{4}$. We will build a platform and an adversary to derive a contradiction. The platform is made up with two processors P_1 and P_2, such that $p_1 = p_2 = p = 2c_2 - c_1$, and $c_1 = \epsilon$, $c_2 = 1$. Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i either on P_1, with a max-flow at least equal to $c_1 + p$, or on P_2 with a max-flow at least equal to $c_2 + p$.

At time step $\tau = c_2 - c_1$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 the adversary send no more task. The best possible max-flow is then $c_2 + p = 3 - \epsilon$. The optimal scheduling being of max-flow $c_1 + p = 2$, we have a competitive ratio of
$$\rho \geq \frac{c_2 + p}{c_1 + p} = \frac{3}{2} - \frac{\epsilon}{2} > \frac{5}{4} - \epsilon$$
Thus the algorithm $\mathcal{A}$ cannot schedule the task i on P_2.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send other tasks. The best max-flow that can be achieved is then equal to $\frac{\tau + c_1 + p}{c_1 + p} = \frac{3-\epsilon}{2}$, which is the same than the previous case. Consequently, the algorithm $\mathcal{A}$ does not have any choice but to schedule the task i on P_1.

At time-step τ, the adversary sends three tasks, j, k and l. No schedule which executes three of the four tasks on the same processor can have a max-flow lower than $max(c_1 + 3p - \tau, max(c_1, \tau) + c_1 + p + \max\{c_1, p\} - \tau) = 6 - 2\epsilon$. We now consider the schedules which compute two tasks on each processor. Since i is computed on P_1, we have three cases to study, depending upon which other task (j, k, or l) is computed on P_1:

1. If j is computed on P_1:
   (a) Task i is sent to P_1 and achieved a flow of $c_1 + p = 2$.
   (b) Task j is sent to P_1 and achieved a flow of $\max\{c_1 + 2p - \tau, \max\{\tau, c_1\} + c_1 + p - \tau\} = 3$.
   (c) Task k is sent to P_2 and achieved a flow of $\max\{\tau, c_1\} + c_1 + c_2 + p - \tau = 3$
   (d) Task l is sent to P_2 and achieved a flow of $\max\{\tau, c_1\} + c_1 + c_2 + p + max\{c_2, p\} - \tau = 5 - \epsilon$.

   The max-flow of this schedule is then $5 - \epsilon$.

2. If k is computed on P_1:
   (a) Task i is sent to P_1 and achieved a flow of $c_1 + p = 2$.
   (b) Task j is sent to P_2 and achieved a flow of $\max\{\tau, c_1\} + c_2 + p - \tau = 3 - \epsilon$.
   (c) Task k is sent to P_1 and achieved a flow of $\max\{c_1 + 2p, \max\{\tau, c_1\} + c_2 + c_1 + p\} - \tau = 3$.
   (d) Task l is sent to P_2 and achieved a flow of $\max\{\max\{\tau, c_1\} + c_2 + 2p, \max\{\tau, c_1\} + 2c_2 + c_1 + p\} - \tau = 5 - 2\epsilon$.

   The max-flow of this scheduling is then $5 - 2\epsilon$.

3. If l is computed on P_1:
   (a) Task i is sent to P_1 and achieved a flow of $c_1 + p = 2$.

(b) task j is sent to P_2 and achieved a flow of $\max\{\tau, c_1) + c_2 + p\} = 3 - \epsilon$.

(c) Task k is sent to P_2 and achieved a flow of $\max\{\max\{\tau, c_1\} + c_2 + 2p, \max\{\tau, c_1\} + 2c_2 + p\} = 5 - 2\epsilon$.

(d) Task l is sent to P_1 and achieved a flow of $\max\{c_1+2p, \max\{\tau, c_1\}+2c_2+c_1+p\}-\tau = 4$.

The max-flow of this schedule is then $5 - 2\epsilon$.

Consequently, the last two schedules are equivalent and are better than the first. Altogether, the best achievable max-flow is $5 - 2\epsilon$. But a better schedule is obtained when computing i on P_2, then j on P_1, then k on P_2, and finally l on P_1. The max-flow of the latter schedule is equal to $\max\{c_2+p, \max\{\tau, c_2\}+c_1+p-\tau, \max\{\max\{\tau, c_2\}+c_1+c_2+p, c_2+2p\}-\tau, \max\{\max\{\tau, c_2\}+2c_1+c_2+p, \max\{\tau, c_2\}+c_1+2p\}-\tau\} = 4$. The competitive ratio of algorithm $\mathcal{A}$ is necessarily larger than the ratio of the best reachable max-flow (namely $5 - 2\epsilon$) and the optimal max-flow, which is not larger than 4. Consequently:

$$\rho \geq \frac{5 - 2\epsilon}{4} = \frac{5}{4} - \frac{\epsilon}{2}$$

which contradicts the assumption $\rho = \frac{5}{4} - \epsilon$ with $\epsilon > 0$.

■

Theorem A.6. *There is no scheduling algorithm for the problem*

$$P_{>1}, MS \mid online,\ r_i,\ p_j = p,\ c_j \mid \sum (C_i - r_i)$$

whose competitive ratio ρ is strictly lower than $23/22$.

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = 23/22 - \epsilon$, with $\epsilon > 0$. We will build a platform and an adversary to derive a contradiction. The platform is made up with two processors P_1 and P_2, such that $p_1 = p_2 = p = 3$, and $c_1 = 1$, $c_2 = 2$. Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i either on P_1, with a max-flow at least equal to $c_1 + p$, or on P_2 with a max-flow at least equal to $c_2 + p$.

At time step $\tau = c_2$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 the adversary sends no more task. The best possible sum-flow is then $c_2 + p = 5$. The optimal scheduling being of sum-flow $c_1 + p = 4$, we have a competitive ratio of
$$\rho \geq \frac{c_2 + p}{c_1 + p} = \frac{5}{4} > \frac{23}{22}.$$
Thus the algorithm $\mathcal{A}$ cannot schedule the task i on P_2.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send other tasks. The best sum-flow that can be achieved is then equal to $\frac{\tau+c_1+p}{c_1+p} = \frac{6}{4}$, which is even worse than the previous case. Consequently, the algorithm $\mathcal{A}$ does not have any choice but to schedule the task i on P_1.

At time-step τ, the adversary sends three tasks, j, k, and l. We look at all the possible schedules, with i computed on P_1:

1. If all tasks are executed on P_1 the sum-flow is $(c_1+p)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_1+p-\tau)+(\max\{c_1+3p,\max\{\tau,c_1\}+c_1+p+\max\{c_1,p\}-\tau)+(\max\{c_1+4p,\max\{\tau,c_1\}+c_1+p+2\max\{c_1,p\}-\tau)=28$.
2. If j is the only task executed on P_2 the sum-flow is $(c_1+p)+(\max\{\tau,c_1\}+c_2+p-\tau)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_2+c_1+p\}-\tau)+(\max\{c_1+3p,\max\{\tau,c_1\}+c_2+c_1+p+\max\{c_1,p\}\}-\tau)=24$.
3. If k is the only task executed on P_2 the sum-flow is $(c_1+p)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_1+p-\tau)+(\max\{\tau,c_1\}+c_1+c_2+p-\tau)+(\max\{c_1+3p,\max\{\tau,c_1\}+c_2+2c_1+p\}-\tau)=23$.
4. If l is the only task executed on P_2 the sum-flow is $(c_1+p)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_1+p-\tau)+(\max\{c_1+3p,\max\{\tau,c_1\}+c_1+p+\max\{c_1,p\}-\tau)+(\max\{\tau,c_1\}+2c_1+c_2+p-\tau)=24$.
5. If j,k,l are executed on P_2 the sum-flow is $(c_1+p)+(\max\{\tau,c_1\}+c_2+p-\tau)+(\max\{\tau,c_1\}+c_2+p+\max\{c_2,p\}-\tau)+(\max\{\tau,c_1\}+c_2+p+2\max\{c_2,p\}-\tau)=28$.

We now consider the schedules which compute two tasks on each processor. Since i is computed on P_1, we have three cases to study, depending upon which other task (j, k, or l) is computed on P_1:

1. If j is computed on P_1 the sum-flow is: $(c_1+p)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_1+p-\tau)+(\max\{\tau,c_1\}+c_1+c_2+p-\tau)+(\max\{\tau,c_1\}+c_1+c_2+p+\max\{c_2,p\}-\tau)=24$.
2. If k is computed on P_1: $(c_1+p)+(\max\{\tau,c_1\}+c_2+p-\tau)+(\max\{c_1+2p,\max\{\tau,c_1\}+c_2+c_1+p\}-\tau)+(\max\{\tau,c_1\}+c_2+p+\max\{c_1+c_2,p\}-\tau)=23$.
3. If l is computed on P_1: $(c_1+p)+(\max\{\tau,c_1\}+c_2+p-\tau)+(\max\{\tau,c_1\}+c_2+p+\max\{c_2,p\}-\tau)+(\max\{c_1+2p,\max\{\tau,c_1\}+2c_2+c_1+p\}-\tau)=25$.

Consequently, the best achievable sum-flow is 23. But a better schedule is obtained when computing i on P_2, then j on P_1, then k on P_2, and finally l on P_1. The sum-flow of the latter schedule is equal to $(c_2+p)+(\max\{\tau,c_2\}+c_1+p-\tau)+(\max\{\max\{\tau,c_2\}+c_1+c_2+p,c_2+2p\}-\tau+\max\{\max\{\tau,c_2\}+2c_1+c_2+p,\max\{\tau,c_2\}+c_1+2p\}-\tau\}=22$. The competitive ratio of algorithm $\mathcal{A}$ is necessarily larger than the ratio of the best reachable sum-flow (namely 23) and the optimal sum-flow, which is not larger than 22. Consequently:

$$\rho \geq \frac{23}{22}$$

which contradicts the assumption $\rho = \frac{23}{22} - \epsilon$ with $\epsilon > 0$. ■

Heterogeneous platforms

In this section, we consider fully heterogeneous platforms.

Theorem A.7. *There is no scheduling algorithm for the problem*

$$Q_{>2}, MS \mid online,\ r_i,\ p_j,\ c_j \mid \max(\mathcal{C}_i - r_i)$$

whose competitive ratio ρ is strictly lower than $\sqrt{2}$.

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \sqrt{2}-\epsilon$, with $\epsilon > 0$. We will build a platform and an adversary to derive a contradiction. The platform is made up of three processors P_1, P_2, and P_3 such that $p_1 = \epsilon$, $p_2 = p_3 = \sqrt{2}c_1 - 1$, $c_1 = 2(1+\sqrt{2})$, $c_2 = c_3 = 1$, and $\tau = (\sqrt{2}-1)c_1$. Note that $\tau < c_1$, and that $c_1 + p_1 < p_2$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i, either on P_1 with a max-flow at least equal to $c_1 + p_1 = c_1 + \epsilon$, or on P_2 or P_3, with a max-flow at least equal to $c_2 + p_2 = c_3 + p_3 = \sqrt{2}c_1$.

At time-step τ, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 or P_3, the adversary does not send any other task. The best possible max-flow is then $c_2 + p_2 = c_3 + p_3 = \sqrt{2}c_1$. The optimal scheduling being of max-flow $c_1 + p_1 = c_1 + \epsilon$, we have a competitive ratio of:

$$\rho \geq \frac{c_2 + p_2}{c_1 + p_1} = \frac{\sqrt{2}c_1}{c_1 + \epsilon} > \sqrt{2} - \epsilon,$$

   as $c_1 > \sqrt{2}$. This contradicts the hypothesis on ρ. Thus the algorithm $\mathcal{A}$ cannot schedule task i on P_2 or P_3.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send any other task. The best max-flow that can be achieved is then equal to $\tau + c_1 + p_1 = \sqrt{2}c_1 + \epsilon$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have any other choice than to schedule task i on P_1.

Then, at time-step τ, the adversary sends two tasks, j and k. We consider all the scheduling possibilities:

- j and k are scheduled on P_1. Then the best achievable max-flow is:

$$\begin{aligned}\max \left\{ \begin{array}{l} c_1 + p_1, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1, c_1 + 2p_1\} - \tau, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1 + \max\{c_1, p_1\}, c_1 + 3p_1\} - \tau \end{array} \right. & \\ = 3c_1 + p_1 - \tau & \\ = (4 - \sqrt{2})c_1 + \epsilon &\end{aligned}$$

  as $p_1 < c_1$.

- The first of the two jobs, j and k, to be scheduled is scheduled on P_2 (or P_3) and the other one on P_1. Then, the best achievable max-flow is:

$$\begin{aligned}\max \left\{ \begin{array}{l} c_1 + p_1, \\ (\max\{c_1, \tau\} + c_2 + p_2) - \tau, \\ \max\{\max\{c_1, \tau\} + c_2 + c_1 + p_1, c_1 + 2p_1\} - \tau \end{array} \right. & \\ = c_1 + c_2 - \tau + \max\{p_2, c_1 + p_1\} & \\ = 2c_1 &\end{aligned}$$

- The first of the two jobs j and k to be scheduled is scheduled on P_1 and the other one on P_2 (or P_3). Then, the best achievable max-flow is:

$$\max \begin{cases} c_1 + p_1, \\ \max\{\max\{c_1, \tau\} + c_1 + p_1, c_1 + 2p_1\} - \tau, \\ (\max\{c_1, \tau\} + c_1 + c_2 + p_2) - \tau\} \end{cases}$$
$$= 2c_1 - \tau + \max\{p_1, c_2 + p_2\}$$
$$= 3c_1$$

- One of the jobs j and k is scheduled on P_2 and the other one on P_3.

$$\max\{c_1 + p_1, (\max\{c_1, \tau\} + c_2 + p_2) - \tau, (\max\{c_1, \tau\} + c_2 + c_3 + p_3) - \tau\}$$
$$= c_1 + 2c_2 + p_2 - \tau$$
$$= 2c_1 + 1$$

- The case where j and k are both executed on P_2, or both on P_3, leads to an even worse max-flow than the previous case. Therefore, we do not need to study it.

Therefore, the best achievable max-flow for $\mathcal{A}$ is: $2c_1$. However, we could have scheduled i on P_2, j on P_3, and then k on P_1, thus achieving a max-flow of:

$$\max\{c_2 + p_2, (\max\{c_2, \tau\} + c_3 + p_3) - \tau, (\max\{c_2, \tau\} + c_3 + c_1 + p_1) - \tau\}$$
$$= \max\{c_2, \tau\} + c_2 + \max\{p_2, c_1 + p_1\} - \tau$$
$$= \sqrt{2}c_1$$

Therefore, we have a competitive ratio of:

$$\rho \geq \frac{2c_1}{\sqrt{2}c_1} = \sqrt{2},$$

which contradicts the hypothesis on ρ. ■

Theorem A.8. *There is no scheduling algorithm for the problem*

$$Q_{>2}, MS \mid \textit{online},\ r_i,\ p_j,\ c_j \mid \sum(\mathcal{C}_i - r_i)$$

whose competitive ratio ρ is strictly lower than $\frac{\sqrt{13}-1}{2}$.

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \frac{\sqrt{13}-1}{2} - \epsilon$, with $\epsilon > 0$. We will build a platform and an adversary to derive a contradiction. The platform is made up of three processors P_1, P_2, and P_3 such that $p_1 = \epsilon$, $p_2 = p_3 = \tau + c_1 - 1$, $c_2 = c_3 = 1$, and $\tau = \frac{\sqrt{52c_1^2+12c_1+1}-(6c_1+1)}{4}$. Note that $\tau < c_1$ and that:

$$\lim_{c_1 \to +\infty} \frac{\tau}{c_1} = \frac{\sqrt{13} - 3}{2}$$

Therefore the exists a value N_0 such that:

$$c_1 \geq N_0 \quad \Rightarrow \quad c_1 > \epsilon \text{ and } \tau > \epsilon.$$

The value of c_1 will be defined later on. For now, we just assume that $c_1 \geq N_0$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i, either on P_1 with a sum-flow at least equal to $c_1 + p_1$, or on P_2 or P_3, with a sum-flow at least equal to $c_2 + p_2 = c_3 + p_3 = \tau + c_1$.

At time-step τ, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 or P_3, the adversary does not send any other task. The best possible sum-flow is then $c_2 + p_2 = c_3 + p_3 = \tau + c_1$. The optimal scheduling being of sum-flow $c_1 + p_1 = c_1 + \epsilon$, we have a competitive ratio of:

$$\rho \geq \frac{\tau + c_1}{c_1 + \epsilon}.$$

However,

$$\lim_{c_1 \to +\infty} \frac{\tau + c_1}{c_1 + \epsilon} = \frac{\frac{\sqrt{13}-3}{2} c_1 + c_1}{c_1} = \frac{\sqrt{13} - 1}{2}.$$

Therefore, there exists a value N_1 such that:

$$c_1 \geq N_1 \quad \Rightarrow \quad \frac{\tau + c_1}{c_1 + \epsilon} > \frac{\sqrt{13} - 1}{2} - \frac{\epsilon}{2},$$

which contradicts the hypothesis on ρ. We will now suppose that $c_1 \geq \max\{N_0, N_1\}$. Then the algorithm $\mathcal{A}$ cannot schedule task i on P_2 or P_3.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send any other task. The best sum-flow that can be achieved is then equal to $\tau + c_1 + p_1 = \tau + c_1 + \epsilon$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have any other choice than to schedule task i on P_1.

Then, at time-step τ, the adversary sends two tasks, j and k. We consider all the scheduling possibilities:

- Tasks j and k are scheduled on P_1. Then the best achievable sum-flow is:

$$\begin{aligned}(c_1 + p_1) + (&\max\{\max\{c_1, \tau\} + c_1 + p_1, c_1 + 2p_1\} - \tau) \\ &+ (\max\{\max\{c_1, \tau\} + c_1 + p_1 + \max\{c_1, p_1\}, c_1 + 3p_1\} - \tau) \\ &= 6c_1 + 3p_1 - 2\tau \\ &= 6c_1 - 2\tau + 3\epsilon\end{aligned}$$

as $p_1 < c_1$.

- The first of the two jobs, j and k, to be scheduled is scheduled on P_2 (or P_3) and the other one on P_1. Then, the best achievable sum-flow is:

$$\begin{aligned}(c_1 + p_1) + ((&\max\{c_1, \tau\} + c_2 + p_2) - \tau) \\ &+ (\max\{\max\{c_1, \tau\} + c_2 + c_1 + p_1, c_1 + 2p_1\} - \tau) \\ &= 4c_1 + 2 + 2p_1 + p_2 - 2\tau \\ &= 5c_1 - \tau + 1 + 2\epsilon\end{aligned}$$

- The first of the two jobs j and k to be scheduled is scheduled on P_1 and the other one on P_2 (or P_3). Then, the best achievable sum-flow is:

$$\begin{aligned}(c_1+p_1)+(\max\{\max\{c_1,\tau\}+c_1+p_1,c_1+2p_1\}-\tau)&\\ +((\max\{c_1,\tau\}+c_1+c_2+p_2)-\tau)&\\ =5c_1+c_2+2p_1+p_2-2\tau&\\ =6c_1-\tau+2\epsilon&\end{aligned}$$

- One of the jobs j and k is scheduled on P_2 and the other one on P_3.

$$\begin{aligned}(c_1+p_1)+((\max\{c_1,\tau\}+c_2+p_2)-\tau)+((\max\{c_1,\tau\}+c_2+c_3+p_3)-\tau)&\\ =3c_1+3c_2+p_1+2p_2-2\tau&\\ =5c_1+1+\epsilon&\end{aligned}$$

- The case where j and k are both executed on P_2, or both on P_3, leads to an even worse sum-flow than the previous case. Therefore, we do not need to study it.

Therefore, the best achievable sum-flow for $\mathcal{A}$ is: $5c_1-\tau+1+2\epsilon$ (as $c_1>\tau>\epsilon$). However, we could have scheduled i on P_2, j on P_3, and then k on P_1, thus achieving a sum-flow of:

$$\begin{aligned}(c_2+p_2)+((\max\{c_2,\tau\}+c_3+p_3)-\tau)+((\max\{c_2,\tau\}+c_3+c_1+p_1)-\tau)&\\ =c_1+3c_2+p_1+2p_2&\\ =3c_1+2\tau+1+\epsilon.&\end{aligned}$$

Therefore, we have a competitive ratio of:

$$\rho\geq\frac{5c_1-\tau+1+2\epsilon}{3c_1+2\tau+1+\epsilon}$$

However,

$$\lim_{c_1\to+\infty}\frac{5c_1-\tau+1+2\epsilon}{3c_1+2\tau+1+\epsilon}=\lim_{c_1\to+\infty}\frac{5c_1-\frac{\sqrt{13}-3}{2}c_1}{3c_1+2\frac{\sqrt{13}-3}{2}c_1}=\frac{\sqrt{13}-1}{2}$$

Therefore, there exists a value N_2 such that:

$$c_1\geq N_2\quad\Rightarrow\quad\frac{5c_1-\tau+1+2\epsilon}{3c_1+2\tau+1+\epsilon}>\frac{\sqrt{13}-1}{2}-\frac{\epsilon}{2},$$

which contradicts the hypothesis on ρ.

Therefore, if we initially choose c_1 greater than $\max\{N_0,N_1,N_2\}$, we obtain the desired contradiction. ■

Appendix B

Matrix product detailed experimental results

This appendix details all the results obtained during the heterogeneous MPI experiments of Section 4.8.4.

Heterogeneous memory sizes

Matrix size	8000 x 64000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	1972	1.00	6	11832	1.61	2668	1.20	6	16008	1.73
HomI	2323	1.18	5	11615	1.58	2605	1.17	5	13025	1.41
Het	2076	1.05	6	12456	1.69	2220	1.00	7	15540	1.68
BMM	2410	1.22	8	19280	2.62	2645	1.19	8	21160	2.29
ODDOML	2171	1.10	8	17368	2.36	2289	1.03	8	18312	1.98
OMMOML	3685	1.87	2	7370	1.00	4614	2.08	2	9228	1.00
ORROML	2351	1.19	8	18808	2.55	2717	1.22	8	21736	2.36

Matrix size	8000 x 96000 x 8000					8000 x 112000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	2953	1.19	6	17718	1.59	3660	1.19	6	21960	1.73
HomI	3291	1.33	5	16455	1.48	3637	1.18	6	21822	1.72
Het	2483	1.00	7	17381	1.56	3215	1.05	7	22505	1.77
BMM	3215	1.29	8	25712	2.31	4011	1.31	8	32088	2.52
ODDOML	2498	1.01	8	19984	1.79	3073	1.00	8	24584	1.93
OMMOML	5570	2.24	2	11140	1.00	6356	2.07	2	12714	1.00
ORROML	2887	1.16	8	23096	2.07	3133	1.02	8	25064	1.97

Matrix size	8000 x 124000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work
Hom	3913	1.19	6	23478	1.59
HomI	3916	1.19	6	23496	1.59
Het	3507	1.06	6	21042	1.42
BMM	4237	1.28	8	33896	2.29
ODDOML	3298	1.00	8	26384	1.79
OMMOML	7384	2.24	2	14774	1.00
ORROML	3925	1.19	8	31400	2.13

ALGO	Throughput				
LP	11258				
Hom	4057	3748	4064	3825	4089
HomI	3444	3839	3646	3849	4086
Het	4848	4505	4833	4355	4562
BMM	3320	3781	3733	3490	3776
ODDOML	4837	4369	4804	4556	4851
OMMOML	2170	2167	2154	2202	2166
ORROML	4364	3681	4157	4469	4076

Matrix size	8000 x 64000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	3340	1.35	2	6680	1.00	4186	1.41	2	8370	1.00
HomI	2481	1.00	4	9924	1.49	3275	1.10	4	13100	1.57
Het	2492	1.00	5	12455	1.86	2974	1.00	5	14870	1.78
BMM	4216	1.70	8	33728	5.05	5665	1.90	8	45320	5.41
ODDOML	3197	1.29	8	25576	3.83	4013	1.35	8	32096	3.83
OMMOML	2481	1.00	5	12405	1.86	3350	1.13	5	16750	2.00
ORROML	3140	1.27	8	25120	3.76	3967	1.33	8	31736	3.79

Matrix size	8000 x 96000 x 8000					8000 x 112000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	5052	1.38	2	10104	1.00	5844	1.39	2	11688	1.00
HomI	3673	1.00	4	14692	1.45	4505	1.07	4	18020	1.54
Het	3783	1.03	5	18910	1.87	4202	1.00	5	21010	1.80
BMM	6318	1.72	8	50544	5.00	7867	1.87	8	62936	5.38
ODDOML	4994	1.36	8	39952	3.95	5347	1.27	8	42776	3.66
OMMOML	3769	1.03	5	18845	1.87	4439	1.06	5	22195	1.90
ORROML	4363	1.19	8	34904	3.45	4890	1.16	8	39120	3.35

Matrix size	8000 x 124000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work
Hom	6709	1.37	2	13416	1.00
HomI	4897	1.00	4	19588	1.46
Het	5135	1.05	5	25675	1.91
BMM	8762	1.79	8	70096	5.22
ODDOML	6325	1.29	8	50600	3.77
OMMOML	5085	1.04	5	25425	1.90
ORROML	6281	1.28	8	50248	3.75

ALGO	Throughput				
LP	6388				
Hom	2395	2389	2375	2395	2385
HomI	3224	3053	3267	3107	3267
Het	3211	3362	3172	3331	3115
BMM	1897	1765	1899	1779	1826
ODDOML	2502	2492	2402	2618	2529
OMMOML	3224	2985	3183	3153	3146
ORROML	2547	2520	2750	2863	2547

Heterogeneous computation capabilities

Matrix size	8000 x 64000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	3199	1.58	8	25592	3.85	3933	1.38	8	31464	3.78
HomI	2588	1.28	6	15528	2.34	3347	1.17	6	20082	2.41
Het	2022	1.00	6	12132	1.83	2853	1.00	6	17118	2.06
BMM	2345	1.16	8	18760	2.82	3900	1.37	8	31200	3.75
ODDOML	3114	1.54	8	24912	3.75	3244	1.14	8	25952	3.12
OMMOML	3322	1.64	2	6644	1.00	4164	1.46	2	8328	1.00
ORROML	3109	1.54	8	24872	3.74	3936	1.38	8	31488	3.78

Matrix size	8000 x 96000 x 8000					8000 x 112000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	4683	1.35	8	37464	3.71	4870	1.35	8	38960	3.34
HomI	3525	1.02	6	21150	2.10	4365	1.21	6	26190	2.25
Het	3461	1.00	8	27688	2.74	3615	1.00	8	28920	2.48
BMM	4019	1.16	8	32152	3.19	4208	1.16	8	33664	2.89
ODDOML	3491	1.01	8	27928	2.77	3726	1.03	8	29808	2.56
OMMOML	5044	1.46	2	10088	1.00	5825	1.61	2	11650	1.00
ORROML	4685	1.35	8	37480	3.72	4878	1.35	8	39024	3.35

Matrix size	8000 x 124000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work
Hom	6283	1.63	8	50264	3.78
HomI	5129	1.33	6	30774	2.32
Het	4026	1.04	8	32208	2.42
BMM	4414	1.14	8	35312	2.66
ODDOML	3862	1.00	8	30896	2.33
OMMOML	6642	1.72	2	13284	1.00
ORROML	6227	1.61	8	49816	3.75

ALGO	Throughput				
LP	7234				
Hom	2501	2543	2562	2875	2547
HomI	3091	2988	3404	3207	3120
Het	3956	3505	3467	3873	3974
BMM	3412	2564	2986	3327	3625
ODDOML	2569	3083	3437	3757	4143
OMMOML	2408	2401	2379	2403	2409
ORROML	2573	2541	2561	2870	2569

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	4811	1.50	7	33677	2.13	8932	1.49	7	62524	4.21
HomI	4742	1.48	7	33194	2.10	8957	1.49	7	62699	4.22
Het	3649	1.14	8	29192	1.85	6708	1.12	6	40248	2.71
BMM	3984	1.24	8	31872	2.02	6965	1.16	8	55720	3.75
ODDOML	3203	1.00	8	25624	1.62	6011	1.00	8	48088	3.24
OMMOML	3945	1.23	4	15780	1.00	7422	1.23	2	14844	1.00
ORROML	4834	1.51	8	38672	2.45	9219	1.53	8	73752	4.97

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	3704	1.24	4	14816	1.06	2768	1.05	4	11072	1.05
HomI	3507	1.18	4	14028	1.00	2989	1.13	4	11956	1.13
Het	2979	1.00	6	17874	1.27	2642	1.00	4	10568	1.00
BMM	3854	1.29	8	30832	2.20	6013	2.28	8	48104	4.55
ODDOML	3112	1.04	8	24896	1.77	4250	1.61	8	34000	3.22
OMMOML	5102	1.71	3	15306	1.09	3221	1.22	6	19326	1.83
ORROML	3083	1.03	8	24664	1.76	3916	1.48	8	31328	2.96

ALGO	Throughput			
LP	6689	3426	6476	6432
Hom	2078	1120	2700	3613
HomI	2109	1117	2852	3346
Het	2740	1491	3357	3785
BMM	2510	1436	2595	1663
ODDOML	3122	1664	3213	2353
OMMOML	2534	1347	1960	3105
ORROML	2068	1085	3244	2554

Fully heterogeneous platforms

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	4485	1.62	6	26910	2.37	7914	1.50	6	47484	4.49
HomI	4990	1.80	5	24950	2.19	5285	1.00	2	10570	1.00
Het	2773	1.00	6	16638	1.46	5737	1.09	5	28685	2.71
BMM	4109	1.48	8	32872	2.89	7672	1.45	8	61376	5.81
ODDOML	3190	1.15	8	25520	2.24	7453	1.41	8	59624	5.64
OMMOML	5688	2.05	2	11376	1.00	6949	1.31	2	13898	1.31
ORROML	5224	1.88	8	41792	3.67	8964	1.70	8	71712	6.78

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	9138	1.81	6	54828	2.04	5252	1.41	7	36764	3.76
HomI	7131	1.41	5	35655	1.32	4889	1.31	2	9778	1.00
Het	5058	1.00	7	35406	1.32	3723	1.00	6	22338	2.28
BMM	6314	1.25	8	50512	1.88	5248	1.41	8	41984	4.29
ODDOML	5424	1.07	8	43392	1.62	4229	1.14	8	33832	3.46
OMMOML	8973	1.77	3	26919	1.00	4359	1.17	4	17436	1.78
ORROML	6732	1.33	8	53856	2.00	4596	1.23	8	36768	3.76

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	6563	1.41	5	32815	2.62	6233	1.46	5	31165	2.89
HomI	6263	1.35	2	12526	1.00	5452	1.27	5	27260	2.52
Het	4647	1.00	5	23235	1.85	4282	1.00	6	25692	2.38
BMM	6705	1.44	8	53640	4.28	5880	1.37	8	47040	4.35
ODDOML	4677	1.01	8	37416	2.99	4443	1.04	8	35544	3.29
OMMOML	8020	1.73	2	16040	1.28	10802	2.52	1	10802	1.00
ORROML	5890	1.27	8	47120	3.76	6320	1.48	8	50560	4.68

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	5790	1.39	7	40530	3.21	10472	1.74	6	62832	3.30
HomI	6309	1.52	2	12618	1.00	10056	1.67	6	60336	3.17
Het	4151	1.00	6	24906	1.97	6027	1.00	7	42189	2.22
BMM	5676	1.37	8	45408	3.60	7308	1.21	8	58464	3.07
ODDOML	4583	1.10	8	36664	2.91	6381	1.06	8	51048	2.68
OMMOML	4943	1.19	4	19772	1.57	19014	3.15	1	19014	1.00
ORROML	7427	1.79	8	59416	4.71	10397	1.73	8	83176	4.37

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	5736	1.28	7	40152	2.86	7921	1.45	7	55447	4.12
HomI	4674	1.04	3	14022	1.00	5780	1.06	5	28900	2.14
Het	4488	1.00	5	22440	1.60	5468	1.00	7	38276	2.84
BMM	5957	1.33	8	47656	3.40	5828	1.07	8	46624	3.46
ODDOML	5016	1.12	8	40128	2.86	5602	1.02	8	44816	3.33
OMMOML	5175	1.15	3	15525	1.11	13474	2.46	1	13474	1.00
ORROML	5942	1.32	8	47536	3.39	9564	1.75	8	76512	5.68

Matrix size	8000 x 80000 x 8000					8000 x 80000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	4908	1.17	7	34356	1.74	6610	1.41	5	33050	1.97
HomI	5128	1.22	6	30768	1.56	6022	1.29	5	30110	1.80
Het	4284	1.02	7	29988	1.52	4672	1.00	6	28032	1.67
BMM	5027	1.19	8	40216	2.04	6678	1.43	8	53424	3.19
ODDOML	4212	1.00	8	33696	1.71	5095	1.09	8	40760	2.43
OMMOML	6585	1.56	3	19755	1.00	8371	1.79	2	16742	1.00
ORROML	7042	1.67	8	56336	2.85	5293	1.13	8	42344	2.53

Matrix size	8000 x 320000 x 8000					8000 x 320000 x 8000				
ALGO	Makespan	Relative perf.	# procs	Work	Relative work	Makespan	Relative perf.	# procs	Work	Relative work
Hom	10867	1.44	6	65202	1.00	7711	1.11	11	84821	1.06
HomI	10842	1.44	6	65052	1.00	7720	1.11	11	84920	1.06
Het	7799	1.04	10	77990	1.20	7268	1.05	11	79948	1.00
BMM	10980	1.46	20	219600	3.38	10180	1.47	20	203600	2.55
ODDOML	7531	1.00	20	150620	2.32	6944	1.00	20	138880	1.74
OMMOML	12207	1.62	6	73242	1.13	7718	1.11	11	84898	1.06
ORROML	8862	1.18	20	177240	2.72	7015	1.01	20	140300	1.75

ALGO	Throughput													
LP	5818	5954	4428	6202	4923	6028	6188	3990	6914	5675	6074	4269	10202	13564
Hom	2230	1264	1094	1904	1524	1604	1727	955	1743	1262	2037	1513	3681	5187
HomI	2004	1892	1402	2046	1597	1834	1585	994	2139	1730	1950	1661	3689	5181
Het	3606	1743	1977	2686	2152	2335	2409	1659	2228	1829	2334	2140	5129	5504
BMM	2434	1303	1584	1905	1491	1701	1762	1368	1679	1716	1989	1497	3643	3929
ODDOML	3135	1342	1844	2365	2138	2251	2182	1567	1994	1785	2374	1963	5311	5760
OMMOML	1758	1439	1114	2294	1247	926	2023	526	1932	742	1519	1195	3277	5183
ORROML	1914	1116	1485	2176	1698	1582	1346	962	1683	1046	1420	1889	4514	5702

Appendix C

From theoretical throughput to realistic schedule

In this section, we briefly explain how the optimality result of Section 5.4.2 for the model **BMP-FC-SS** can be adapted to the other models. As expected, the more realistic the model, the less tight the optimality guaranty. Fortunately, we are always able to reach *asymptotic optimality*: our schedules get closer to the optimal as the number of tasks per application increases.

Property of the one-dimensional load-balancing schedule

First, we need to introduce a tool that will prove helpful for the proofs: the one-dimensional load-balancing schedule and its properties.

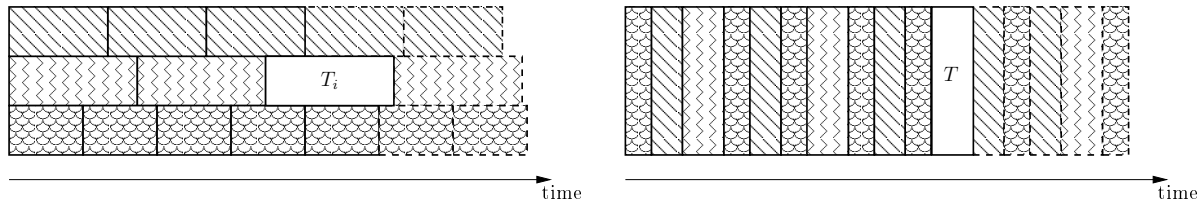
One part of Chapter 5 is devoted to comparing results under different models. One of the major differences between these models is whether they allow –or not– preemption and time-sharing. On the one hand, we study "fluid" models, where a resource (processor or communication link) can be simultaneously used by several tasks, provided that the total utilization rate is below one. On the other hand, we also study "atomic" models, where a resource can be devoted to only one task, which cannot be preempted: once a task is started on a given resource, this resource cannot perform other tasks before the first one is completed. In this section, we show how to construct a schedule without preemption from fluid schedules, in a way that keeps the interesting properties of the original schedule. Namely, we aim at constructing atomic-model schedules in which tasks terminate not later, or start not earlier, than in the original fluid schedule.

We consider a general case of n applications $A_1, \ldots, A_n$ to be scheduled on the same resource, typically a given processor, and we denote by t_k the time needed to process one task of application A_k at full speed. We start from a fluid schedule S_{fluid} where each application A_k is processed at a rate of α_k tasks per time-units, such that $\sum_{k=1}^{n} \alpha_k \leq 1$. Figure C.2(a) illustrates such a schedule.

From S_{fluid}, we build an atomic-model schedule S_{1D} using a one-dimensional load-balancing algorithm [36, 12]: at any time step, if n_k is the number of tasks of application A_k that have already been scheduled, the next task to be scheduled is the one which minimizes the quantity $\frac{(n_k+1)\times t_k}{\alpha_k}$. Figure C.2(b) illustrates the schedule obtained. We now prove that this schedule has the nice property that a task is not processed later in S_{1D} than in S_{fluid}.

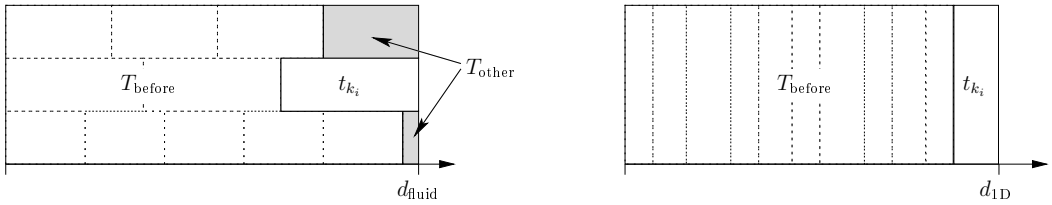
Lemma C.1. *In the schedule S_{1D}, a task T does not terminate later than in S_{fluid}.*

Figure C.1: Gantt charts for the proof illustrating the one-dimensional load-balancing algorithm.



(a) fluid schedule S_{fluid} (b) atomic schedule $S_{1\text{D}}$

Proof. First, we point out that t_k/α_k is the time needed to process one task of application A_k in S_{fluid} (with rate α_k). So $\frac{n_k \times t_k}{\alpha_k}$ is the time needed to process the first n_k tasks of application A_k. The scheduling decision which chooses the application minimizing $\frac{(n_k+1)\times t_k}{\alpha_k}$ consists in choosing the task which is not yet scheduled and which terminates first in S_{fluid}. Thus, in $S_{1\text{D}}$, the tasks are executed in the order of their termination date in S_{fluid}. Note that if several tasks terminate at the very same time in S_{fluid}, then these tasks can be executed in any order in $S_{1\text{D}}$, and the partial order of their termination date is still observed in $S_{1\text{D}}$.



Then, consider a task T_i of a given application A_{k_i}, its termination date d_{fluid} in S_{fluid}, and its termination date $d_{1\text{D}}$ in $S_{1\text{D}}$. We call $\mathcal{S}_{\text{before}}$ the set of tasks which are executed before T_i in $S_{1\text{D}}$. Because $S_{1\text{D}}$ executes the tasks in the order of their termination date in S_{fluid}, $\mathcal{S}_{\text{before}}$ is made of tasks which are completed before T_i in S_{fluid}, and possibly some tasks completed at the same time as T_i (at time d_{fluid}). We denote by T_{before} the time needed to process the tasks in $\mathcal{S}_{\text{before}}$.

In $S_{1\text{D}}$, we have $d_{1\text{D}} = T_{\text{before}} + t_{k_i}$ whereas in S_{fluid}, we have $d_{\text{fluid}} = T_{\text{before}} + t_{k_i} + T_{\text{other}}$ where T_{other} is the time spent processing tasks from other application than A_k and which are not completed at time d_{fluid}, or tasks completing at time d_{fluid} and scheduled later than T_i in $S_{1\text{D}}$. Since $T_{\text{other}} \geq 0$, we have $d_{1\text{D}} \leq d_{\text{fluid}}$. ■

The previous property is useful when we want to construct an atomic-model schedule, that is a schedule without preemption, in which task results are available no later than in a fluid schedule. On the contrary, it can be useful to ensure that no task will start earlier in an atomic-model schedule than in the original fluid schedule. Here is a procedure to construct a schedule with the latter property.

1. We start again from a fluid schedule S_{fluid}, of makespan M. We transform this schedule into a schedule S^{-1}_{fluid} by reversing the time: a task starting at time d and finishing at time f in S_{fluid} is scheduled to start at time $M - f$ and to terminate at $M - d$ in S^{-1}_{fluid}, and is processed at the same rate as in S_{fluid}. Note that this is possible since we have no precedence constraints between tasks.

2. Then, we apply the previous one-dimensional load-balancing algorithm on S_{fluid}^{-1}, leading to the schedule S_{1D}^{-1}. Thanks to the previous result, we know that a task T does not terminate later in S_{1D}^{-1} than in S_{fluid}^{-1}.

3. Finally, we transform S_{1D}^{-1} by reverting the time one last time: we obtain the schedule S_{1D}^{-2}. A task starting at time d and finishing at time f in S_{1D}^{-1} starts at time $M - f$ and finishes at time $M - d$ in S_{1D}^{-2}. Note that S_{1D}^{-1} may have a makespan smaller that M (if the resource was not totally used in the original schedule S_{fluid}). In this case, our method automatically introduces idle time in the one-dimensional schedule, to avoid that a task is started too early.

Lemma C.2. *A task does not start sooner in S_{1D}^{-2} than in S_{fluid}.*

Proof. Consider a task T, call f_1 its termination date in S_{fluid}^{-1}, and f_2 its termination date in S_{1D}^{-1}. Thanks to Lemma C.1, we know that $f_2 \leq f_1$. By construction of the reverted schedules, the starting date of task T in S_{fluid} is $M - f_1$. Similarly, its starting date in S_{1D}^{-2} is $M - f_2$ and we have $M - f_2 \geq M - f_1$. ■

Quasi-optimality for more realistic models

Here we explain how the optimality result of Section 5.4.2 can be adapted to the other models. We describe the delay induced by each model in comparison to the fluid model: starting from a schedule optimal under the fluid model (BMP-FC-SS), we try to build a schedule with comparable performance under a more constrained scenario.

In the following, we consider a schedule S_1, with stretch $\mathcal{S}$, valid under the totally fluid model (BMP-FC-SS). For the sake of simplicity, we consider that this schedule has been built from a point in Polyhedron (K) as explained in section 5.4.2: the computation and communication rates ($\rho_u^{(k)}(t_j, t_{j+1})$ and $\rho_{M\to u}^{(k)}(t_j, t_{j+1})$) are constant during each interval, and are defined by the coordinates of the point in Polyhedron (K).

We assess the *delay* induced by each model. Given the stretch $\mathcal{S}$, we can compute a deadline $d^{(k)}$ for each application A_k. By moving to more constrained models, we will not be able to ensure that the finishing time $MS^{(k)}$ is smaller than $d^{(k)}$. We call lateness for application A_k the quantity $\max\{0, MS^{(k)} - d^{(k)}\}$, that is the time between the due date of an application and its real termination. Once we have computed the maximum lateness for each model, we show how to obtain asymptotic optimality.

Without simultaneous start: the BMP-FC model

We consider here the BMP-FC model, which differs from the previous model only by the fact that a task cannot start before it has been totally received by a processor.

Theorem C.1. *From schedule S_1, we can build a schedule S_2 obeying the BMP-FC model where the maximum lateness for each application is* $\max_{1\leq u\leq p} \sum_{k=1}^{n} \frac{w^{(k)}}{s_u^{(k)}}$.

Proof. From the schedule S_1, valid under the fluid model (BMP-FC-SS), we aim at building S_2 with a similar stretch where the execution of a task cannot start before the end of the corresponding communication. We first build a schedule as follows, for each processor P_u ($1 \leq u \leq p$):

1. Communications to P_u are the same as in S_1;
2. By comparison to S_1, the computations on P_u are shifted for each application A_k: the computation of the first task of A_k is not really performed (P_u is kept idle instead of computing this task), and we replace the computation of task i by the computation of task $i-1$.

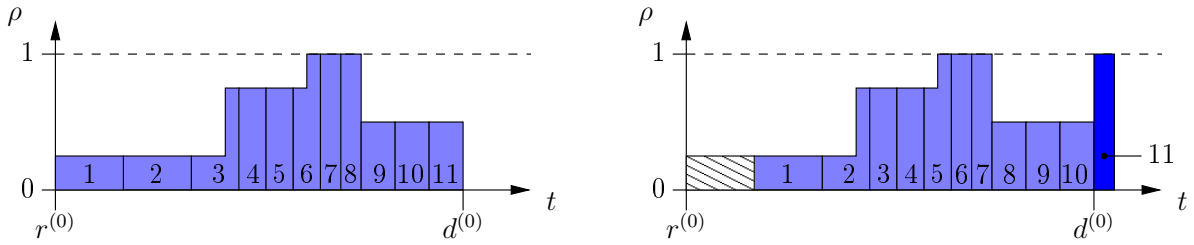
Because of the shift of the computations, the last task of application A_k is not executed in this schedule at time $d^{(k)}$. We complete the construction of S_2 by adding some delay after deadline $d^{(k)}$ to process this last task of application A_k at full speed, which takes a time $\frac{w^{(k)}}{s_u^{(k)}}$. All the following computations on processor P_u (in the next time-intervals) are shifted by this delay.

The lateness for any application A_k on processor P_u is at most the sum of the delays for all applications on this processor, $\sum_{k=1}^{n} \frac{w^{(k)}}{s_u^{(k)}}$, and the total lateness of A_k is bounded by the maximum lateness between all processors:

$$lateness^{(k)} \leq \max_{1 \leq u \leq p} \sum_{k=1}^{n} \frac{w^{(k)}}{s_u^{(k)}}$$

An example of such a schedule S_2 is shown in Figure C.2 (on a single processor). ■

Figure C.2: Example of the construction of a schedule S_2 for BMP-FC model from a schedule S_1 for BMP-FC-SS model. We plot only the computing rate. Each box corresponds to the execution of one task.



(a) Schedule S_1 (BMP-FC-SS model)

(b) Schedule S_2 (BMP-FC model)

Atomic execution of tasks: the BMP-AC model

We now move to the BMP-AC model, where a given processor cannot compute several tasks in parallel, and the execution of a task cannot be preempted: a started task must be completed before any other task can be processed.

Theorem C.2. *From schedule S_1, we can build a schedule S_3 obeying the BMP-AC model where the maximum lateness for each application is*

$$\max_{1 \leq u \leq p} 2n \times \sum_{k=1}^{n} \frac{w^{(k)}}{s_u^{(k)}}.$$

Proof. Starting from a schedule S_1 valid under the fluid model (BMP-FC-SS), we want to build S_3, valid in BMP-AC. We take here advantage of the properties of one-dimensional load-balancing schedules, and especially of $S_{1\mathrm{D}}^{-2}$. Schedule S_3 is built as follows:

1. Communications are kept unchanged;
2. We consider the computations taking place in S_1 on processor P_u during time-interval $[t_j, t_{j+1}]$. A rational number of tasks of each application may be involved in the fluid schedule. We first compute the integer number of tasks of application A_k to be computed in S_3:
$$n_{u,j,k} = \left\lfloor \rho_u^{(k)}(t_j, t_{j+1}) \times (t_{j+1} - t_j) \right\rfloor.$$
The first $n_{u,j,k}$ tasks of A_k scheduled in time-interval $[t_j, t_{j+1}]$ on P_u are organized using the transformation to build $S_{1\mathrm{D}}^{-2}$ in the one-dimensional load-balancing section.
3. Then, the computations are shifted as for S_2: for each application A_k, the computation of the first task of A_k is not really performed (the processor is kept idle instead of computing this task), and we replace the computation of task i by the computation of task $i-1$.

Lemma C.2 proves that, during time-interval $[t_j, t_{j+1}]$, on processor P_u, a computation does not start earlier in S_3 than in S_1. As S_1 obeys the totally fluid model (BMP-FC-SS), a computation of S_1 does not start earlier than the corresponding communication, so a computation of task i of application A_k in S_1 does not start earlier than the finish time of the communication for task $i-1$ of A_k. Together with the shifting of the computations, this proves that in S_3, the computation of a task does not start earlier than the end of the corresponding communication, on each processor.

Because of the rounding down to the closest integer, on each processor P_u, at each time-interval, S_3 computes at most one task less than S_1 of application A_k. Moreover, one more task computation of application A_k is not performed in S_3 due to the computation shift. On the whole, as there are at most $2n-1$ time-intervals, at most $2n$ tasks of A_k remain to be computed on P_u at time $d^{(k)}$. The delay for application A_k is:

$$lateness^{(k)} \leq \max_{1 \leq u \leq p} 2n \times \sum_{k=1}^{n} \times \frac{w^{(k)}}{s_u^{(k)}}.$$

■

This is obviously not the most efficient way to construct a schedule for the BMP-AC model: in particular, each processor is idle during each interval (because of the rounding down). It would certainly be more efficient to sometimes start a task even if it cannot be terminated before the end of the interval. This is why for the experiments of section 5.4.4, we implemented on each worker a greedy schedule with Earliest Deadline First Policy instead of this complex construction. However, we can easily prove that this construction has an asymptotic optimal stretch, unlike other greedy strategies.

Asymptotic optimality

In this section, we show that the previous schedules are close to the optimal, when applications are composed of a large number of tasks. To establish such an asymptotic optimality, we have to prove that the gap computed above gets smaller when the number of tasks gets larger. At

first sight, we would have to study the limit of the application stretch when $\Pi^{(k)}$ is large for each application. However, if we simply increase the number of tasks in each application without changing the release dates and the tasks characteristics, then the problem will look totally different: any schedule will run for a very long time, and the time separating the release dates will be negligible in front of the whole duration of the schedule. This behavior is not meaningful for our study.

To study the asymptotic behavior of the system, we rather change the granularity of the tasks: we show that when applications are composed of a large number of small-size tasks, then the maximal stretch is close to the optimal one obtained with the fluid model. To take into account the application characteristics, we introduce the granularity g, and we redefine the application characteristics with this new variable:

$$\Pi_g^{(k)} = \frac{\Pi^{(k)}}{g}, \qquad w_g^{(k)} = g \times w^{(k)} \quad \text{and} \quad \delta_g^{(k)} = g \times \delta^{(k)}.$$

When $g = 1$, we get back to the previous case. When $g < 1$, there are more tasks but they have smaller communication and computation size. For any g, the total communication and computation amount per application is kept the same, thus it is meaningful to consider the original release dates.

Our goal is to study the case $g \to 0$. Note that under the totally fluid model (BMP-FC-SS), the granularity has no impact on the performance (or the stretch). Indeed, the fluid model can be seen as the extreme case where $g = 0$. The optimal stretch under the BMP-FC-SS $\mathcal{S}_{\text{opt}}$ does not depend on g.

Theorem C.3. *When the granularity is small, the schedule constructed above for the BMP-FC (respectively BMP-AC) model is asymptotically optimal for the maximum stretch, that is*

$$\lim_{g \to 0} \mathcal{S} = \mathcal{S}_{\text{opt}}$$

where $\mathcal{S}$ is the stretch of the BMP-FC (resp. BMP-AC) schedule, and $\mathcal{S}_{\text{opt}}$ the stretch of the optimal fluid schedule.

Proof. The lateness of the applications computed for the BMP-FC model, and for the BMP-AC model, becomes smaller when the granularity increase: for the BMP-FC model, we have

$$\mathit{lateness}^{(k)} \leq \max_{1 \leq u \leq p} \sum_{k=1}^{n} \frac{w_g^{(k)}}{s_u^{(k)}} \xrightarrow[g \to 0]{} 0.$$

Similarly, for the BMP-AC model,

$$\mathit{lateness}^{(k)} \leq \max_{1 \leq u \leq p} 2n \times \sum_{k=1}^{n} \frac{w_g^{(k)}}{s_u^{(k)}} \xrightarrow[g \to 0]{} 0.$$

Thus, when g gets close to 0, the stretch obtained by these schedules is close to $\mathcal{S}_{\text{opt}}$. ∎

One-port model

In this section, we explain how to modify the previous study to cope with the one-port model. We cannot simply extend the result obtained for the fluid model to the one-port model (as we

have done for the other models) since the parameters for modeling communications are not the same. Actually, the one-port model limits the time spent by a processor (here the master) to send data whereas the multiport model limits its bandwidth capacity. Thus, we have to modify the corresponding constraints. Constraint (5.10) of Section 5.4.2 is replaced by the following one.

$$\forall 1 \leq j \leq 2n-1, \sum_{u=1}^{p} \sum_{k=1}^{n} \rho_{M\to u}^{(k)}(t_j, t_{j+1}) \frac{\delta^{(k)}}{b_u} \leq 1 \qquad (5.10\text{-b})$$

Note that the only difference with Constraint (5.10) is that, now, we bound the time needed by the master to send all data instead of the volume of the data itself. The set of constraints corresponding to the scheduling problem under the one-port model, for a maximum stretch $\mathcal{S}$, are gathered by the definition of Polyhedron (K_1):

$$\begin{cases} \rho_{M\to u}^{(k)}(t_j, t_{j+1}), \rho_u^{(k)}(t_j, t_{j+1}), \ \forall k, u, j \text{ such that } 1 \leq k \leq n, 1 \leq u \leq p, 1 \leq j \leq 2n-1 \\ \text{under the constraints (5.3), (5.7), (5.5), (5.6), (5.4), (5.8), (5.9), (5.10-b), and (5.11)} \end{cases} \qquad (K_1)$$

As in Section 5.4.2, the existence of a point in the polyhedron is linked to the existence of a schedule with stretch $\mathcal{S}$. However, we have no fluid model which could perfectly follow the behavior of the linear constraints. Thus we only target asymptotic optimality.

Theorem C.4.

(a) *If there exists a schedule valid under the one-port model with stretch $\mathcal{S}_1$, then Polyhedron (K_1) is not empty for $\mathcal{S}_1$.*

(b) *Conversely, if Polyhedron (K_1) is not empty for the stretch objective $\mathcal{S}_2$, then there exists a schedule valid for the problem under the one-port model with parameters $\Pi_g^{(k)}$, $\delta_g^{(k)}$, and $w_g^{(k)}$, as defined previously, whose stretch $\mathcal{S}$ is such that*

$$\lim_{g\to 0} \mathcal{S} = \mathcal{S}_2.$$

Proof. **(a)** To prove the first part of the theorem, we prove that for any schedule with stretch $\mathcal{S}_1$, we can construct a point in Polyhedron (K_1). Given such a schedule, we denote by $A_{M\to u}^{(k)}(t_j, t_{j+1})$ the total number of tasks of application A_k sent by the master to processor P_u during interval $[t_j, t_{j+1}]$. Note that this may be a rational number if there are ongoing transfers at times t_j and/or t_{j+1}. Similarly, we denote by $A_u^{(k)}(t_j, t_{j+1})$ the total (rational) number of tasks of A_k processed by P_u during interval $[t_j, t_{j+1}]$. Then we compute:

$$\rho_{M\to u}^{(k)}(t_j, t_{j+1}) = \frac{A_{M\to u}^{(k)}(t_j, t_{j+1})}{t_{j+1} - t_j} \quad \text{and} \quad \rho_u^{(k)}(t_j, t_{j+1}) = \frac{A_u^{(k)}(t_j, t_{j+1})}{t_{j+1} - t_j}.$$

As in the fluid case, we can also compute the state of the buffers based on these quantities:

$$B_u^{(k)}(t_j) = \sum_{t_i+1 \leq t_j} A_{M\to u}^{(k)}(t_i, t_{i+1}) - A_u^{(k)}(t_i, t_{i+1})$$

We can easily check that all constraints (5.3),(5.4), (5.5), (5.6), (5.7), (5.8), (5.9), and (5.10-b) are satisfied. Variables $B_u^{(k)}(t_j)$, $\rho_{M\to u}^{(k)}(t_j, t_{j+1})$, and $\rho_u^{(k)}(t_j, t_{j+1})$ define a point in Polyhedron (K_1).

(b) From a point in Polyhedron (K_1), we build a schedule which is asymptotically optimal, as defined in the previous section. During each interval $[t_j, t_{j+1}]$, for each worker P_u, we proceed as follows.

1. We first consider a fluid-model schedule S_f following exactly the rates defined by the point in the polyhedron: the tasks of application A_k are sent with rate $\rho^{(k)}_{M\to u}(t_j, t_{j+1})$ and processed at rate $\rho^{(k)}_u(t_j, t_{j+1})$.
2. We transform both the communication schedule and the computation schedule using one-dimensional load-balancing algorithms. We first compute the integer number of tasks that can be sent in the one-port schedule:

$$n^{\text{comm}}_{u,j,k} = \left\lfloor \rho^{(k)}_{M\to u}(t_j, t_{j+1}) \times (t_{j+1} - t_j) \right\rfloor .$$

   The number of tasks that can be computed on P_u in this time-interval is bounded both by the number of tasks processed in the fluid-model schedule, and by the number of tasks received during this time-interval plus the number of remaining tasks:

$$n^{\text{comp}}_{u,j,k} = \min \left\{ \left\lfloor \rho^{(k)}_u(t_j, t_{j+1}) \times (t_{j+1} - t_j) \right\rfloor \ , \ n^{\text{comm}}_{u,j,k} + \sum_{i=1}^{j-1} \left(n^{\text{comm}}_{u,i,k} - n^{\text{comp}}_{u,i,k} \right) \right\}$$

   The first $n^{\text{comm}}_{u,j,k}$ tasks sent in schedule S_f are organized with the one-dimensional load-balancing algorithm into $S_{1\text{D}}$, while the last $n^{\text{comp}}_{u,j,k}$ tasks executed in schedule S_f are organized with the inverse one-dimensional load-balancing algorithm $S^{-2}_{1\text{D}}$.
3. Then, the computations are shifted: for each application A_k, the computation of the first task of A_k is not really performed (the processor is kept idle instead of computing this task), and we replace the computation of task i by the computation of task $i-1$.

The proof of the validity of the obtained schedule is very similar to the proof of Theorem C.2 for the BMP-AC model: we use the fact that a task does not start earlier in $S^{-2}_{1\text{D}}$ than in S_f, and no later in $S_{1\text{D}}$ than in S_f to prove that the data needed for the execution of a given task are received in time.

At time $d^{(k)}$, some tasks of application A_k are still not processed, and some may even not be received yet. Let us denote by L_k the number of time-intervals between $r^{(k)}$ and $d^{(k)}$, that is time-intervals where tasks of application A_k may be processed ($L_k \leq 2n-1$). Because of the rounding of the numbers of tasks sent, at most one task is not transmitted in each interval, for each application. At time $d^{(k)}$, we thus have at most L_k tasks of application A_k to be sent to each processor P_u. We have to serialize the sending operations, which takes a time at most

$$\sum_{u=1}^{n} \frac{L_k \times \delta^{(k)}}{b_u}$$

Then, the number of tasks remaining to be processed on processor P_u is upper bounded by $2L_k + 1$: at most L_k are received late because of the rounding of the number of tasks received, at most L_k tasks are received but not computed because we also round the number of tasks processed, and one more task may also remain because of the computation shift. The computation (at full speed) of all these tasks takes at most a time $(2L_k+1)\frac{w^{(k)}}{s^{(k)}_u}$ on

processor P_u. Overall, the delay induced on all processors for finishing application A_k can be bounded by:

$$\sum_{u=1}^{n} \frac{L_k \times \delta^{(k)}}{b_u} + \max_{1 \leq u \leq p} (2L_k + 1) \times \frac{w^{(k)}}{s_u^{(k)}}.$$

As $L_k \leq 2n - 1$, the lateness of any application A_k is thus:

$$lateness^{(k)} \leq \sum_{k} \left(\sum_{u=1}^{n} \frac{(2n-1) \times \delta^{(k)}}{b_u} + \max_{1 \leq u \leq p} (4n - 1) \times \frac{w^{(k)}}{s_u^{(k)}} \right).$$

As in the proof of Theorem C.3, when the granularity becomes small, the stretch of the obtained schedule becomes as close to $\mathcal{S}_2$ as we want. ■

Appendix D

Bibliography

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *Proceedings of the 15th annual ACM symposium on Parallelism in algorithms and architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[2] Susanne Albers. On randomized online scheduling. In *Proceedings of the 34th annual ACM symposium on Theory of computing (STOC'02)*, pages 134 – 143. ACM Press, 2002.

[3] Susanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1-2):3–26, 2003.

[4] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Trans. Algorithms*, 3(4):49, 2007.

[5] Hakan Aydin, Rami Melhem, Daniel Mosse, and Pedro Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. *ecrts*, 00:0225, 2001.

[6] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[7] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*. IEEE Computer Society Press, 1998.

[8] M. Banikazemi, J. Sampathkumar, S. Prabhu, D.K. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *HCW'1999, the 8th Heterogeneous Computing Workshop*, pages 125–133. IEEE Computer Society Press, 1999.

[9] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[10] N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic speed scaling to manage energy and temperature. *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 520–529, 17-19 Oct. 2004.

[11] Amotz Bar-Noy, Sudipto Guha, Joseph (Seffi) Naor, and Baruch Schieber. Message multicasting in heterogeneous networks. *SIAM Journal on Computing*, 30(2):347–358, 2000.

[12] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Transactions on Computers*, 50(10):1052–1070, 2001.

[13] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.

[14] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2002.

[15] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Partitioning a square into rectangles: NP-completeness and approximation algorithms. *Algorithmica*, 34:217–239, 2002.

[16] Olivier Beaumont, Vincent Boudet, and Yves Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *HCW'2002, the 11th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2002.

[17] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2006.

[18] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. *IEEE Transactions on Parallel and Distributed Systems*, 19, 2008, to appear.

[19] Olivier Beaumont, Henri Casanova, Arnaud Legrand, Yves Robert, and Yang Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):207–218, 2005.

[20] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Independent and divisible tasks scheduling on heterogeneous star-schaped platforms with limited memory. In *13th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'05)*, pages 179–186. IEEE Computer Society Press, 2005.

[21] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Steady-state scheduling on heterogeneous clusters. *International Journal of Foundations of Computer Science*, 16(2):163–194, 2005.

[22] Olivier Beaumont, Arnaud Legrand, and Yves Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *ISCIS XVII, 17th International Symposium On Computer and Information Sciences*, pages 115–119. CRC Press, 2002.

[23] Olivier Beaumont, Arnaud Legrand, and Yves Robert. The master-slave paradigm with heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):897–908, 2003.

[24] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th annual ACM-SIAM symposium on Discrete Algorithms (SODA'98)*, pages 270–279. Society for Industrial and Applied Mathematics, 1998.

[25] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Approximation algorithms for average stretch scheduling. *Journal of Scheduling*, 7(3):195–222, 2004.

[26] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[27] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[28] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithms for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[29] V. Bharadwaj, D. Ghose, and V. Mani. Multi-installment load distribution in tree networks with delays. *IEEE Transactions on Aerospace Electronic Systems*, 31:555–567, April 1995.

[30] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 15–24. IEEE Computer Society Press, 1999.

[31] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[32] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed-memory computers - design issues and performance. In *Proceedings of the ACM/IEEE Symposium on Supercomputing*. IEEE Computer Society Press, 1996.

[33] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[34] J. Blazewicz, J.K. Lenstra, and A.H. Kan. Scheduling subject to resource constraints. *Discrete Applied Mathematics*, 5:11–23, 1983.

[35] BOINC: Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu.

[36] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25:547–568, 1999.

[37] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[38] David P. Bunde. Power-aware scheduling for makespan and flow. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 190–196, New York, NY, USA, 2006. ACM.

[39] David P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures (SPAA'06)*, pages 190–196, New York, NY, USA, 2006.

[40] T. Burd. *Energy-Efficient Processor System Design*. PhD thesis, 2001. Available at the url http://bwrc.eecs.berkeley.edu/Publications/2001/THESES/energ_eff_process-sys_des/BurdPhD.pdf.

[41] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[42] H. Casanova and F. Berman. Grid'2002. In F. Berman, G. Fox, and T. Hey, editors, *Parameter sweeps on the grid with APST*. Wiley, 2002.

[43] H. Casanova and F. Berman. *Grid Computing: Making The Global Infrastructure a Reality*, chapter Parameter Sweeps on the Grid with APST. John Wiley, 2003. Hey, A. and Berman, F. and Fox, G., editors.

[44] Henri Casanova. Network modeling issues for grid application scheduling. *International Journal of Foundations of Computer Science*, 6(2):145–162, 2005.

[45] A.J. Chakravarti, G. Baumgartner, and M. Lauria. Self-organizing scheduling on the organic grid. *International Journal of High Performance Computing Applications*, 20(1):115–130, 2006.

[46] Ho-Leung Chan, Wun-Tat Chan, Tak-Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. Energy efficient online deadline scheduling. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 795–804, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[47] Chandra Chekuri and Sanjeev Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings of the 34th annual ACM symposium on Theory of computing (STOC'02)*, pages 297–305. ACM Press, 2002.

[48] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and lapack for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.

[49] Yi-Jen Chiang, Ricardo Farias, Cláudio T. Silva, and Bin Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 59–66, Piscataway, NJ, USA, 2001. IEEE Press.

[50] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, Andy B. Yoo, and Chita R. Das. A comprehensive performance and energy consumption analysis of scheduling alternatives in clusters. *Journal of Supercomputing*, 40(2):159–184, 2007.

[51] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[52] M. Cierniak, M.J. Zaki, and W. Li. Compile-time scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[53] M. Cierniak, M.J. Zaki, and W. Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1997.

[54] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício Alves Barbosa da Silva, Carla Osthoff Barros, and Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'03)*, October 2003.

[55] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[56] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[57] James Cowie, Bruce Dodson, R.-Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Joerg Zayer. A world wide number field sieve factoring record: on to 512 bits. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - Asiacrypt '96*, volume 1163 of *LNCS*, pages 382–394. Springer Verlag, 1996.

[58] P. E. Crandall and M. J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 42–49. IEEE Computer Society Press, 1993.

[59] Javier Cuenca, Luis Pedro Garcia, Domingo Gimenez, and Jack Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *International Conference on Heterogeneous Computing*. IEEE Computer Society Press, 2005.

[60] A. Darte, Y. Robert, and F.Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.

[61] Jack Dongarra, Swen Hammarling, and David Walker. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1-2):49–70, 1997.

[62] Pierre-François Dutot. Complexity of master-slave tasking on heterogeneous trees. *European Journal of Operational Research*, 164(3):690–695, August 2005. Special issue on Recent Advances in Scheduling in Computer and manufacturing Systems (J. Blazewicz, K. Ecker, and D. Trystram editors).

[63] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[64] Entropia. URL: http://www.entropia.com.

[65] Dror G. Feitelson. *Workload Characterization and Modeling Book*. electronic draft, no published yet.

[66] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24, 1998.

[67] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, , and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*. IEEE Computer Society Press, August 2001.

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

[69] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. *Journal of Supercomputing*, 0:34, 2005.

[70] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994.

[71] GLPK: GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[72] J. P Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*. IEEE Computer Society Press, 2000.

[73] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. see also http://www-unix.mcs.anl.gov/mpi/mpich/.

[74] William Gropp. Mpich2: A new start for mpi implementations. In *PVM/MPI*, page 7, 2002.

[75] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

[76] HMMER. URL: http://hmmer.wustl.edu/.

[77] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *Journal of Combinatorial Optimization*, 1(2):129–149, 1997.

[78] B. Hong and V.K. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'03)*. IEEE Computer Society Press, 2003.

[79] B. Hong and V.K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2004.

[80] J.-W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the annual ACM symposium on Theory of computing (STOC'81)*, pages 326–333. ACM Press, 1981.

[81] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Transactions on Computers*, 49(12):1339–1353, 2000.

[82] Dror Ironya, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[83] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202, New York, NY, USA, 1998. ACM.

[84] San Jeddi and Wahid Nasri. A poly-algorithm for efficient parallel matrix multiplication on metacomputing platforms. In *CLUSTER*, pages 1–9, 2005.

[85] M. Kaddoura, S. Ranka, and A. Wang. Array decomposition for nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 36:91–105, 1996.

[86] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.

[87] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing*, 61(4):520–535, 2001.

[88] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[89] S. Khuller and Y.A. Kim. On broadcasting in heterogenous networks. In *Proceedings of the 15th annual ACM symposium on Discrete Algorithms (SODA'04)*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.

[90] B. Kreaseck. *Dynamic autonomous scheduling on Heterogeneous Systems.* PhD thesis, University of California, San Diego, 2003.

[91] Barbara Kreaseck, Larry Carter, Henri Casanova, and Jeanne Ferrante. On the interference of communication on computation in java. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2004.

[92] Barbara Kreaseck, Larry Carter, Henri Casanova, Jeanne Ferrante, and Sagnik Nandy. Interference-aware scheduling. *International Journal of High Performance Computing Applications*, 20(1):45–59, 2006.

[93] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2004.

[94] A. Legrand, L.Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, May 2003.

[95] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Mapping and load-balancing iterative computations on heterogeneous clusters with shared links. *IEEE Trans. Parallel Distributed Systems*, 15(6):546–558, 2004.

[96] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of biological requests. In *Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures (SPAA'06)*, pages 103–112, Cambridge, Massachusetts, USA, 2006. ACM Press.

[97] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. Research report rr-6002, INRIA, 2006. http://hal.inria.fr/inria-00108524. Also available as LIP research report RR2006-19.

[98] J.K. Lenstra, R. Graham, E. Lawler, and A.H. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[99] Joseph Y-T. Leung and Hairong Zhao. Minimizing mean flowtime and makespan on master-slave systems. *Journal of Parallel and Distributed Computing*, 65(7):843–856, 2005.

[100] Joseph Y-T. Leung and Hairong Zhao. Minimizing sum of completion times and makespan in master-slave systems. *Transactions on Computers*, 55(8):985–999, August 2006.

[101] Keqin Li and Victor Y. Pan. Parallel matrix multiplication on a linear array with a reconfigurable pipelined bus system. *IEEE Transactions on Computers*, 50(5):519–525, 2001.

[102] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111. IEEE Computer Society Press, 1988.

[103] P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.

[104] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *HCW'1999, the 8th Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

[105] J.M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1), September 1968.

[106] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes Gehrke. Online scheduling to minimize average stretch. In *IEEE Symposium on Foundations of Computer Science*, pages 433–442, 1999.

[107] W. Nasri and D. Trystram. A polyalgorithmic approach applied for fast matrix multiplication on clusters. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 234–, April 2004.

[108] The network simulator ns 2. URL: http://www.isi.edu/nsnam/ns.

[109] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):103–117, 1993.

[110] Takanori Okuma, Tohru Ishihara, and Hiroto Yasuura. Real-time task scheduling for a variable voltage processor. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 24, Washington, DC, USA, 1999. IEEE Computer Society.

[111] J. M. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 349–354. IEEE Computer Society Press, 2001.

[112] Jean-François Pineau, Yves Robert, and Frédéric Vivien. The impact of heterogeneity on master-slave on-line scheduling. In *HCW'2006, the 15th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2006.

[113] Jean-François Pineau, Yves Robert, and Frédéric Vivien. Off-line and on-line scheduling on heterogeneous master-slave platforms. In *PDP'2006, 14th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, Montbéliard-Sochaux, France, February 15-17 2006. IEEE Computer Society Press.

[114] Jean-François Pineau, Yves Robert, Frédéric Vivien, and Jack Dongarra. Matrix product on heterogeneous master-worker platforms. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, 2008.

[115] Mencoder Media Player. URL: http://www.mplayerhq.hu/.

[116] C. D. Polychronopoulos. Compiler optimization for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1988.

[117] Prime. URL: http://www.mersenne.org.

[118] Kirk Pruhs, Jiří Sgall, and Eric Torng. On-line scheduling. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 15.1–15.43. Chapman & Hall/CRC Press, 2004.

[119] Gang Quan and Xiaobo Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference*, pages 828–833, 2001.

[120] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[121] C. Roig, A. Ripoll, M. A. Senar, F. Guirado, and E. Luque. Improving static scheduling using inter-task concurrency measures. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 375–381. IEEE Computer Society Press, 2001.

[122] Cosmin Rusu, Rami Melhem, and Daniel Mossé. Multi-version scheduling in rechargeable energy-aware real-time systems. *J. Embedded Comput.*, 1(2):271–283, 2005.

[123] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182, 2004.

[124] Simone Santini. We are sorry to inform you ... *Computer*, 38(12):128, 126–127, 2005.

[125] Andreas S. Schulz and Martin Skutella. The power of α-points in preemptive single machine scheduling. *Journal of Scheduling*, 5(2):121–133, 2002. DOI:10.1002/jos.093.

[126] SETI. URL: http://setiathome.ssl.berkeley.edu.

[127] J. Sgall. On line scheduling-a survey. In *On-Line Algorithms*, Lecture Notes in Computer Science 1442, pages 196–231. Springer-Verlag, Berlin, 1998.

[128] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources.* PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.

[129] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop (HCW).* IEEE Computer Society Press, 2000.

[130] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[131] Barbara Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, 12(2):294–299, 1983.

[132] O. Sinnen. *Task Scheduling for Parallel Systems.* Wiley, May 2007.

[133] Oliver Sinnen and Leonel Sousa. Experimental evaluation of task scheduling accuracy: Implications for the scheduling mode. *IEICE TRANSACTIONS on Information and Systems*, E86-D(9):1620–1627, 2003. Special Issue on Parallel and Distributed Computing, Applications and Technologies.

[134] Oliver Sinnen and Leonel Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[135] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[136] David Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, 07(4):26–35, 1999.

[137] Wayne E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[138] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):857–871, 1997.

[139] Qinghui Tang, Sandeep. K. S. Gupta, Daniel Stanzione, and Phil Cayton. Thermal-aware task scheduling to minimize energy usage of blade server based datacenters. *dasc*, 00:195–202, 2006.

[140] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[141] Krijn van der Raadt, Yang Yang, and Henri Casanova. Practical divisible load scheduling on grid platforms with apst-dv. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 29.2, Washington, DC, USA, 2005. IEEE Computer Society.

[142] Girish Varatkar and Radu Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 510, Washington, DC, USA, 2003. IEEE Computer Society.

[143] A. Watt. *3D Computer Graphics*. Addison-Wesley Longman Publishing Co., Boston, 1993.

[144] J. B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop (HCW)*. IEEE Computer Society Press, 2000.

[145] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Symposium on Supercomputing*. IEEE Computer Society Press, 1998.

[146] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[147] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Energy minimization of real-time tasks on variable voltage processors with transition energy overhead. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 65–70, New York, NY, USA, 2003. ACM.

[148] Dakai Zhu, Rami 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*, 14(7):686–700, 2003.

[149] Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):433–448, 2007.

Appendix E

Publications

The publications are listed in reverse chronological order.

Articles in international refereed journals

[A1] Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. *International Journal of Foundations of Computer Science*, 2008. To appear.

[A2] Jean-François Pineau, Yves Robert, and Frédéric Vivien. The impact of heterogeneity on master-slave scheduling. *Parallel Computing*, 34(3):158–176, 2008.

Articles in international refereed conferences

[B1] Jean-François Pineau, Yves Robert, and Frédéric Vivien. Off-line and on-line scheduling on heterogeneous master-slave platforms. In *PDP'2006, 14th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE Computer Society Press, 2006.

[B2] Jean-François Pineau, Yves Robert, and Frédéric Vivien. The impact of heterogeneity on master-slave on-line scheduling. In *HCW'2006, the 15th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2006.

[B3] Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. In *9th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2007*. IEEE Computer Society Press, 2007.

[B4] Jack Dongarra, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Matrix product on heterogeneous master-worker platforms. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, Utah, February 20-23 2008.

[B5] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Offline and online master-worker scheduling of concurrent bags-of-tasks on heterogeneous platforms. In *10th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2008*. IEEE Computer Society Press, 2008.

Research reports

[C1] Jean-François Pineau, Yves Robert, and Frédéric Vivien. Off-line and on-line scheduling on heterogeneous master-slave platforms. Technical report, LIP, July 2005.

[C2] Jean-François Pineau, Yves Robert, and Frédéric Vivien. The impact of heterogeneity on master-slave on-line scheduling. Research report LIP 2005-51, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, October 2005. Also available as INRIA research report 5732.

[C3] Jean-François Pineau Jack Dongarra, Zhiao Shi Yves Robert, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. Research report, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, November 2006. Also available as INRIA research report 6053.

[C4] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Offline and online scheduling of concurrent bags-of-tasks on heterogeneous platforms. Research report, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, December 2007. Also available as INRIA research report 6401.

Appendix F

Notations

The notations used in this thesis are unfortunately not uniform from one chapter to another, because they depend on our choice of model. The notations are also supposed to be coherent with the refering litterature. Here is a brief overview of all the notations used.

Definition F.1 ($\lfloor \boldsymbol{x} \rfloor$ and $\lceil \boldsymbol{x} \rceil$)**.** *For x in $\mathbb{R}$, lets $\lfloor x \rfloor$ be the integer in $\mathbb{Z}$ verifying:*

$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1,$$

and $\lceil x \rceil$ be the integer in $\mathbb{Z}$ such as:

$$\lceil x \rceil - 1 < x \leq \lceil x \rceil,$$

Here are the various platform and application models under study:

- **BMP-FC-SS** is the Bounded Multiport (Section 2.4.2) with Fluid Computation (Section 2.3.2) and Synchronous Start model (Section 2.3.4);
- **BMP-FC** is the Bounded Multiport (Section 2.4.2) with Fluid Computation (Section 2.3.2) model;
- **BMP-AC** is the Bounded Multiport (Section 2.4.2) with Atomic Computation (Section 2.3.3) model;
- **OP-AC** is the One-Port (Section 2.4.2) with Atomic Computation (Section 2.3.3) model.

We describe our scheduling problems using the notation $\alpha \mid \beta \mid \gamma$:

- $\boldsymbol{\alpha}$ denotes the platform. We use $\mathbf{1}$ when we are working on a single processor, $\boldsymbol{P}$ for platforms with identical processors, $\boldsymbol{Q}$ for platforms with different-speed but uniform processors and $\boldsymbol{R}$ for unrelated processors. We use *MS* to indicate that we work on master-worker platforms.
- $\boldsymbol{\beta}$ denotes all the platform and applications constraints. This is where we indicate if we have a communication-homogeneous platform or a computation-homogeneous platform. We can also precise whether or not we have tasks with release dates or deadlines. If this field does not contain a constrain, for example the presence of release dates, then the problem is supposed to be without release dates.
- $\boldsymbol{\gamma}$ denotes the objective. Along this thesis, we deal with several objective functions:
  - the *makespan* or total execution time $max\ \boldsymbol{C_i}$;

- the *max-flow* or maximum response time $max\ (\boldsymbol{C_i - r_i})$;
- the *sum-flow* $\sum(\boldsymbol{C_i - r_i})$;
- the maximum stretch $max\ \frac{\boldsymbol{C_i - r_i}}{\boldsymbol{MS^{*(i)}}}$;
- the average stretch $\sum \frac{\boldsymbol{C_i - r_i}}{\boldsymbol{MS^{*(i)}}}$.

We always considier the master-worker model. The master is denoted P_0 and the workers $P_1, \ldots, P_m$.

Chapter 3

- $\boldsymbol{m}$ is the number of processors of the platform;
- $\boldsymbol{c_u}$ is the time needed by the master to send a task to P_u;
- $\boldsymbol{p_u}$ is the time needed by P_u to execute a task;
- $\boldsymbol{r_i}$ is the release time of task i;
- $\boldsymbol{C_i}$ denotes the end of the execution of task i under the target schedule.

Chapter 4

We describe the platform using the following notations:

- $\boldsymbol{p}$ is the number of processors of the platform;
- we are computing the product $\mathcal{C} = \mathcal{C} + \mathcal{A} \times \mathcal{B}$. Matrix $\mathcal{A}$ is of size $\boldsymbol{r} \times \boldsymbol{t}$, $\mathcal{B}$ is of size $\boldsymbol{s} \times \boldsymbol{t}$ and $\mathcal{C}$ of size $\boldsymbol{r} \times \boldsymbol{s}$. All three matrices are split into blocks of size $\boldsymbol{q} \times \boldsymbol{q}$;
- $\boldsymbol{w_u}$ is the computation speed of P_u;
- $\boldsymbol{c_u}$ is the communication speed of the the link between the master and P_u;
- $\boldsymbol{m_u}$ represent the number of blocks of matrices of size $q \times q$ that can hold into the memory of P_u.

During the proof of the lower bound on the communication volume, we use the following notation:

- $\boldsymbol{\alpha}$ is the number of buffers used by elements of $\mathcal{A}$;
- $\boldsymbol{\beta}$ is the number of buffers used by elements of $\mathcal{B}$;
- $\boldsymbol{\gamma}$ is the number of buffers used by elements of $\mathcal{C}$.

Our homogeneous algorithm will use:

- $\boldsymbol{\mu^2}$ blocks for elements of $\mathcal{C}$, $\boldsymbol{\mu}$ blocks for elements of $\mathcal{B}$, and 1 block for elements of $\mathcal{A}$, $\mu^2 + \mu + 1 \leq m$;
- $\boldsymbol{\mathfrak{P}}$ processors if the matrix is big enough, and the platform fast enough;
- $\boldsymbol{\mathfrak{Q}}$ otherwise.

Chapter 5

Here are the notations used to describe the applications:

- $\boldsymbol{A^{(k)}}$ represents the k^{th} application;
- $\boldsymbol{w^{(k)}}$ represents the computation need of $A^{(k)}$;
- $\boldsymbol{\delta^{(k)}}$ is the communication volume of $A^{(k)}$;
- $\boldsymbol{\Pi^{(k)}}$ represents the number of independant tasks of $A^{(k)}$;
- $\boldsymbol{r^{(k)}}$ is the release date of $A^{(k)}$;

- $\boldsymbol{\mathcal{S}}$ is the actual stretch that we want to achieve;
- $\boldsymbol{\mathcal{S}_{\text{opt}}}$ is the minimal max-stretch of all applications for our platform;
- $\boldsymbol{d^{(k)}}$ is the deadlines of $A^{(k)}$ that we have to achieve a stretch $\mathcal{S}$;
- $\boldsymbol{t_i}$ represents either a release date or a deadline;
- $\boldsymbol{\mathcal{C}^{(k)}}$ represents the completion time of $A^{(k)}$;
- $\boldsymbol{MS^{(k)}}$ is the flow of $A^{(k)}$;
- $\boldsymbol{MS^{*(k)}}$ is the flow of $A^{(k)}$ if the application was alone on the platform.

We describe the platform using the following notations:

- $\boldsymbol{p}$ represents the number of processors of the platform
- $\boldsymbol{b_u}$ represents the bandwidth available from the master to P_u;
- $\mathbf{BW}$ represents the total outgoing capacity of the master;
- $\boldsymbol{s_u^{(k)}}$ represents the speed of worker P_u to process a task of $A^{(k)}$.

With such notations, the time needed to send a task of $A^{(k)}$ to P_u would be $\dfrac{\delta^{(k)}}{b_u}$, while the time needed to compute it on P_u would be $\dfrac{w^{(k)}}{s_u^{(k)}}$.

We use several notation to express our problem with linear programming:

- $\boldsymbol{\rho_u^{(k)}(t_j, t_{j+1})}$ denotes the computation throughput of worker P_u during time-interval $[t_j, t_{j+1}]$ for application A_k, i.e., the average number of tasks of A_k computed by P_u per time-units;
- $\boldsymbol{\rho_{M\to u}^{(k)}(t_j, t_{j+1})}$ denotes the communication throughput from the master to the worker P_u during time-interval $[t_j, t_{j+1}]$ for application A_k, i.e., the average number of tasks of A_k sent to P_u per time-units;
- $\boldsymbol{B_u^{(k)}(t_j)}$ denotes the (fractional) number of tasks of application A_k stored in a buffer on P_u at time t_j;
- $\boldsymbol{A_u^{(k)}(t_j, t_{j+1})}$ denotes the rational number of tasks of A_k processed by P_u during interval $[t_j, t_{j+1}]$;
- $\boldsymbol{A_{M\to u}^{(k)}(t_j, t_{j+1})}$ denotes the rational number of tasks of application A_k sent by the master to processor P_u during interval $[t_j, t_{j+1}]$.

During the part about energy consumption, we introduce some new notations:

- $\boldsymbol{\mathcal{A}}$ is the unique application that is considered in this part;
- $\boldsymbol{m_u}$ denotes the number of frequencies of P_u;
- $\boldsymbol{P_{u,i}}$ denotes that the processor P_u is running at the ith mode;
- $\boldsymbol{s_{u,i}}$ denotes the speed of processor P_u at the ith mode;
- $\boldsymbol{\rho_{u,i}}$ denotes the throughput of $P_{u,i}$;
- $\boldsymbol{\mathfrak{P}_u}$ denotes the power consumption per time-unit of P_u;
- $\boldsymbol{\mathfrak{K}_{u,i}}$ denotes the instantaneous power consumption of P_u $\left(\dfrac{\mathfrak{P}_{u,i}\ w}{s_{u,i}} = \rho_{u,i}\mathfrak{K}_{u,i}\right)$.

Abstract:
The results summarized in this document deal with the scheduling of independent tasks on large scale master-worker platforms, when realistic communication models are utilized. The contributions of this work are divided into three main parts: 1) Parallel algorithms: we underline the difficulty of scheduling identical independent tasks on heterogeneous master-worker platforms using the one-port communication model. We look at several sources of heterogeneity as well as several objective functions; 2) Matrix product: we compute the total communication volume that is needed for matrix multiplication in the presence of memory constraints and when data is centralized, and we develop a memory layout whose performance is close to the theoretical lower bound on the communication volume. We extend this algorithm for heterogeneous platforms; 3) Scheduling: lots of applications are constituted of a very large number of independent identical tasks. We focus on steady-state, and prove how to minimize the slowdown of one application when several are deployed, and how to minimize power consumption when only one application is present.
Keywords:
Scheduling, independent tasks, communication models, master-worker platform.

Résumé :
Les travaux présentés dans cette thèse portent sur diverses techniques d'ordonnancement de tâches indépendantes pour des plates-formes de type maître-esclaves distribuées à grande échelle, lorsque les temps de communications des tâches sont pris en compte par des modèles réalistes. Les contributions de cette thèse se situent à trois niveaux : 1) Algorithmique Parallèle : nous avons montré la complexité d'ordonnancer des tâches indépendantes sur une plate-forme hétérogène en modélisant les communications avec un modèle un-port, en regardant plusieurs sources d'hétérogénéité et plusieurs fonctions objectives ; 2) Produit de matrices : nous avons calculé la borne théorique du volume de communication minimal nécessaire pour effectuer un produit de matrices dont les données sont centralisées, et où la mémoire des esclaves est limitée, et nous avons défini un algorithme efficace de partage de la mémoire, impliquant un volume de communication proche de la borne théorique. Nous avons ensuite étendu cet algorithme à des plate-formes hétérogènes ; 3) Ordonnancement : dans le cadre d'ordonnancement d'applications constituées d'un très grand nombre de tâches indépendantes et de caractéristiques identiques, nous avons étudié en régime permanent comment minimiser le retard de chaque application lorsqu'elles sont plusieurs à entrer en compétition pour les ressources de calcul, et comment minimiser la consommation de la plate-forme lorsqu'une seule application est déployée.
Mots-clés :
Ordonnancement, tâches indépendantes, modèles de communication, plate-forme maître-esclaves.