Mémoire d'habilitation à diriger des recherches

École normale supérieure de Lyon

On scheduling for distributed heterogeneous platforms

Frédéric Vivien

Version du 19 février 2008.

Contents

Foreword (for the non scientific reader)

The most classical work-related question I am subjected to, in family meetings, is something like "What are you working on?". Once again, and in a vernacular language, I am going to explain what problems I am trying to solve.

I am working in the field of scientific computing. Imagine things like big simulations —where one wants to forecast the weather or the behavior of cars in an accident— or computational biology —a biologist search for the already known proteins which look like the one they have just discovered. So you have this computation "so hugely big" that it would run for months, or even for centuries, on a simple computer (and I am not kidding when I write "centuries"). So, you have these programs that simply cannot be run on a single computer. The idea is just to use several computers simultaneously, having them working together, in *parallel* as we say, to perform the big computation. The underlying principle is obvious: if a task is too demanding for a single entity to complete it, ask several entities to collaborate to its resolution. So, you divide the big computation in smaller computations that you distribute among all the available computers. Of course, it would not be fun if things were so simple. Most of the time, you cannot divide the whole computation the way you want. For instance, somewhere in the computation a sub-result is computed, and this sub-result is reused afterward to compute another sub-result: the two sub-results cannot be computed simultaneously on two different computers as one *depends* on the other. So, we must first define a *model of the application*; this model codes the main characteristics of the computation we are trying to perform in parallel. Of course, our models are never perfect. We suppose that we know beforehand the running time of the different sub-computations but we only have estimates, and these estimates can be more or less far off the actual values. We thus have to deal with the model imperfections; we have to cope with all the existing *uncertainties*. For some problems, we cannot even predict whether some sub-computation is going to take place because, for example, this sub-computation is necessary only if the result of some other computation satisfies this or that property. In such cases, we discover some of the application main characteristics only during the execution. We then have to take decisions *dynamically*, that is, while the application is running. Furthermore, we may have to deal with not just one but with several applications, some of them appearing over time without us knowing that we will have to deal with them.

Well, forget about the imperfections of the model for a while. We take our model, we study it —in a mathematical sort of way— to find out its properties, to know what freedom we have in designing a *schedule*. A schedule states which sub-computation takes place on which computer and at which time. Of course, any schedule is not a good solution: the schedule which runs every sub-computation on the very same processor of the very same computer is not a bright idea. The question is then: what is our *objective* ? what do we want to optimize ? what is our aim ? A simple wish is to obtain the result as quickly as possible. If you own the computers or are alone in the world, this can be a good idea. If several people are sharing a same set of computers you will certainly have to find some compromise between the users. In such a case, people may want that your scheduler guarantee some quality of service. Of course, to ensure some quality of service, or just to ensure that your application is completed as fast as possible, you need some knowledge on the set of available computational resources. You thus need some *model of the platform*. This is not just a description of the computational power of the processors you can use. If you envision to simultaneously use a computer in Strasbourg and one in Lyon to execute some application, you need to know how long it is going to take to send a message from one computer to the other. And if you are also going to use a computer in Grenoble and one in Boston, you need to know whether you can simultaneously have a communication between Strasbourg and Lyon and one between Grenoble and Boston without any interference between the two communications, or whether one communication will impact the other. You thus need a description of the communication links interconnecting the processors, a description that enables to predict the execution time of a set of communications. As for the application model, there are uncertainties on the platform model... In general, we will call the platforms we target "distributed heterogeneous platforms" as all the processors used are not necessarily the same, and as they are not necessarily in the same computer.

In this thesis, I will first make a formal presentation of my view of the research field I just introduced in a vernacular language, that is the scheduling of large scientific computations on distributed heterogeneous platforms. I will give an overview of the applications, platforms, and objectives we may consider, of the issues we may encounter while modeling our problems, and of possible research directions (Chapter 1).

In the following chapters (Chapter 2 to 4), I will give an overview of my work, in order to illustrate what I think are important characteristics of this research domain.

Chapter 1

Introduction

In scientific computing, the problem of scheduling is to decide where and when computations should be executed. To solve such a problem, one must know what must be scheduled, what resources are available, and what objectives should be pursued by the scheduler.

We will thus start by describing the different applications (Section 1.1) and platforms (Section 1.2) we may have to deal with. Then, we will discuss the objectives we could try to optimize (Section 1.3). The modeling of platforms and applications raises several important issues (Section 1.4) which, in turn, open several important research directions (Section 1.5). We will later conclude and describe the rest of this document (Section 1.6).

The objective of this chapter is twofold. First we want to give a broad survey of the objects we may be dealing with, in this field of research. The second objective is to pinpoint some of the problems, weaknesses, or limitations, of most of the current work in this field, and to propose some potential research directions.

1.1 Applications

Obviously, there does not exist an application model which would perfectly describe any 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 now give a quick overview of the main ones and of the main application characteristics.

Some of the applications we will have to schedule will be *parallel*, that is, one can, or must, use simultaneously several computing resources to process them. Sometimes, we will know very few things on these applications, and we will mainly see them as black boxes. We may only know the number of processors an application can run on, and the application execution time as a function of the number of processors used. We will have to schedule such a task, simultaneously, on the right number of processors, and during a contiguous interval of time of sufficient duration. If, because of the way the task is written and independently of the characteristics of the computation platform, the number of processors that can be used to process the task is fixed a priori, i.e., if we have absolutely no freedom for choosing it, the task is called *rigid*. Otherwise, if the number of processors may be freely chosen at runtime by the scheduler, but cannot be modified after the beginning of the task processing, the task is called *moldable* [55]. With *malleable* tasks [55] the scheduler has the freedom to change the number of processors used at any time during the task processing.

We may, however, have far more knowledge on the application structure. Mainly, we may have some knowledge on how the application is built as a composition of (possibly many) sub-tasks. In this scope, one of the simpler models is the *divisible load* model. Under this model, an application can be split into an arbitrary number of chunks, each of them of an arbitrary size, and these chunks can be processed independently[1]. The divisible load theory is especially attractive as this model enables to solve several scheduling problems whose solutions can be expressed by neat closed-form formulas. This model is

[1] Most of the time, people associate the divisible load theory with linear communication and computation times. Such an assumption is not mandatory; for instance see [13] for a study considering latencies. A linear cost model may even lead to inconsistent results, for instance when dealing with multi-installment strategies [C1].

perfectly adapted to some applications, as we will see in Sections 2.1 and 4.2. It is not, however, very general.

A related model is the *bag-of-task* model. In this model an application is made of a given number of independent sub-tasks, and each of these sub-tasks is atomic. 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. Typical bag-of-task applications are applications where the same computation must be performed on independent data [91], independent parameter sets [110] (the so-called parameter-sweep applications [36]), or independent models [73].

The *divisible load* and *bag-of-task* models both correspond to trivially parallel applications, that is, to applications whose sub-tasks can be executed in any order or even concurrently. Under a more general scenario, 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).[2] Such an application will be described by a *directed acyclic graph* (DAG) whose vertices represent the sub-tasks and whose edges represent the dependence relations between the sub-tasks. Some complex applications cannot be represented by DAGs. This is for instance the case when part of the application is iterated, or when the application includes a feed-back loop. These days, instead of scheduling *DAGs*, researchers in the field of Grid computing will rather schedule *workflows*, which is, scientifically speaking, exactly the same thing most of the time (but the latter vocable is far more fashionable). I prefer to restrain the use of *workflow* for cases where the whole DAG is not known beforehand but is discovered during the execution of the application. With this meaning, workflows are dynamically discovered DAGs.

In fact, so far, we have never stated when we gain our knowledge of the application structure: for workflows, we learn of the application structure incrementally and during the application execution. In the simplest model, we know beforehand all the application characteristics and all the applications to be scheduled are available and ready to be scheduled. This is for instance the case when you have a computing platform dedicated to your own usage and when you know what you want to run on it. In other contexts, the applications will arrive over time, each application having its own *arrival date* or *release date*. If all release dates are known from the start, the model is said to be *offline*; if release dates are discovered at the time the applications arrive on the platform, the model is said to be *online*. 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. The different models of task arrivals can obviously be combined with any of the application models. For instance, we can have an online or offline scheme of constant arrivals combined with the *bag-of-task* model. This is equivalent to the model where a *stream* or *flow* of tasks is submitted over time. An example of such an application is the decoding of a video stream (a rare example where we can know beforehand each of the many release dates).

Besides release dates, a task may 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. Among other potential characteristics, the different applications may have different *priorities*, and users may be allowed to place *reservations* on resources.

There is thus a large variety of applications model. The above overview is obviously not exhaustive as one can for instance mix directed acyclic graphs and moldable tasks to represent complex structured applications where a sub-task can be moldable [106]. In the remainder of this chapter, we will call *job* a work submitted to the computing system, and *tasks* the constituting part of a job.

As we will see in Section 2.1, the choice of an application model can have a great impact on our availability to efficiently solve the scheduling problem. The relevance of the chosen model may also have a deep impact on the quality of the study: if the chosen model is too far away from the reality, there will be no correlation between the theory predictions and what is observed in practice, and the whole study will be worthless. This problem is even more acute with platform modeling.

[2]Note that even if two sub-tasks are independent of each other —as in a trivially parallel application— their respective executions may still interfere, for instance if they share some input data. In such a case, there may be some interferences between the communications that may be required to bring the common input data to the processors that must process the two sub-tasks. We will consider in detail this problem in Chapter 3.

1.2 Platforms

The two main components of the considered computing platforms are the computing resources themselves, and the interconnection networks. (The memory and data storage capacities are often overlooked but may also play a great role in our algorithmic problems, as will be illustrated in Section 3.3.)

Computing resources

The trademark of the platforms we consider is the heterogeneity of their computing resources. Such a characteristic is obvious in Grid computing platforms, but can also be found in brand new machines like the new CNRS IBM computing platform [80, 46]. Therefore, we will only consider homogeneous platforms as baseline references, or to stress the impact of heterogeneity on the complexity of problems or on the design of solutions.

Scheduling theory is proposing a classification of the heterogeneity of computing resources in three platform types:

1. The *parallel and identical machines* model denotes 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.
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 any machine (because, for instance, it needs some special libraries or operating system). We will 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 not fit in the main memory of all the different processors, and thus will or 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. Preemption is possible if a task can be temporarily stopped and its execution be resumed later on. If a task can be restarted on another processor, we have preemption and migration. A generalization of preemption for parallel machines is *gang scheduling* [60] where all the processors belonging to a same "partition" are simultaneously switching context to run another set of tasks (this is a method to realize time-sharing on parallel computers running parallel applications). Virtualization is a possible means to realize preemption and migration [72].

As we have seen, the choice of a model of computing resources is rather straightforward: basically, we have three processor models to choose from, and to state whether me consider preemption. We will now see that things are far from being so simple with communication capabilities.

Communication resources

A first series of historical scheduling works was simply not considering at all communications or, to put it in another way, in such works everything was done as if communications were instantaneous (see for instance Chapter 5 of [31]). 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 [A5]).

The next widely used model was the *macro-dataflow* model (for instance see the survey papers [44, 57, 104, 120] 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-dataflow 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 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 the general case. These flaws are far from being only theoretical: the executions times predicted using such a model can be unrelated to the actual execution times [122, B10].

Several models are attempts to realistically represent the limitations on communication capabilities of processors. One of them is the *one-port* model [21, 22]. Under this model, a resource can send at most one message at any time and can receive at most one message at any time. Under the *unidirectional* one-port model a message cannot be sent and one received simultaneously, which is allowed under the *bidirectional* one-port model. Another approach is to use the *bounded multi-port* model, where the communication bandwidth of each resource is bounded but not the number of communications it can simultaneously be involved in [78].

The one-port model is a possible solution for the first problem of the macro-dataflow model. To address the second problem, we need some representation of the communication 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 will not be the case in most of the platforms we will consider. The most common approach is then to model the interconnection network by a graph whose edges, representing the (physical or logical) communication links will be labelled with communication capacities (bandwidths, latencies, etc.). In a generalization of this model, hyper-edges denote network buses [123].

Even when a graph-like representation of interconnection networks has been adopted, there remains to define a model of communication times. The most commonly used model is affine in the size of the transmitted data (other models have been proposed and validated, see [9, Section 1.3] for a discussion of some of them). But even this rather simple model may be hard to instantiate. Indeed, if a processor is communicating while it is computing, the computations may have a significant and negative impact on the available bandwidth [89]. Also, when several communications happen simultaneously on a same physical link the fact that they impact the performance of each other, in other words, the fact that they have to "share" the available bandwidths, and how, depends on the nature of the link [38]. Not speaking of low level details such as the bound on the usable bandwidth defined, independently of the available bandwidth, by the TCP-window size and the latency.

The problem of modeling network resources is thus a tricky one. We will come back to this problem in Section 1.4. It should be emphasized that we are only looking for models to accurately describe the network's behavior for an application point of view. We do not need models of packet-level precision, such as the ones used by the network simulator NS [105].

1.3 Objectives

Metrics based on completion times

The most common objective function in the (parallel) scheduling literature is the *makespan*: the maximum of the tasks' 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*. If all jobs do not have the same importance, one may rather use weighted versions of the above metrics, namely the *maximum-weighted completion time*, or the *average-weighted completion time*.

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 the jobs' release dates into account. The relevant parameter is no longer the date at which a job is completed, but the

time it spends in the system, from its release date to its completion time. 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). Note that the average flow time and average completion time problems have the exact same optimal solutions. Furthermore, any guaranteed algorithm for average flow time is guaranteed with the same bound for the average completion time. The converse, however, is not true, and the two optimizations problems are thus quite different (for instance see the example of Smith's ratio rule in Section 4.5).

These flow-based metrics, like those based on completion times, tend to favor long jobs to the detriment of short ones, as long jobs are more constraining. To overcome this problem, one common approach [42] focuses on the *weighted* flow time, using job weights to offset the bias against short jobs. *Sum weighted flow* and *maximum weighted flow* metrics can then be analogously defined. The *stretch* [15] or *slowdown* [61] is a particular case of weighted flow. On a single processor, a job's weight is inversely proportional to its size. The stretch of a job 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., the scarcity of a particular database as in Chapter 4), we adapt the stretch definition to deal with the unrelated machines context. We thus define the stretch as a particular case of weighted flow, in which a job's weight is inversely proportional to its processing time when the system is empty. In other words, the slowdown is defined as the runtime experienced on the loaded system divided by the runtime that would have been achieved if the system had been dedicated.[3] In order not to give too much importance to very short jobs, such as jobs failing at start time, one may wish to use instead a *bounded slowdown* where the weights cannot be smaller than a fixed lower bound (e.g., corresponding to a 10 second runtime as in [61]).

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. Theorem 10 (on page 71) shows that 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, from sum-completion time to sum weighted flow. In a system where fairness matters, sum-based metrics cannot be used.

In the next paragraph we present a few game theory notions and how they translate to our context. This enables us to understand a major flaw of the max-based metrics: they are only optimizing the most constraining jobs, hence the denomination of *bottleneck objectives* in [31].

Pareto-Optimality

Game theory provides a general framework to model situations where many users compete for resources. Each user (in our context, a job) is characterized by a *utility* function u_j. The utility functions represent the satisfaction perceived by the users (typically function of the delay or of the capacity). The goal is to find scheduling strategies such that the utility of *each* user is maximized. In our context it is more relevant to consider cost functions rather than utility functions. Indeed, scheduling problems are typically minimization problems as we try to minimize the completion time, the flow or the stretch of each job (we will therefore assume in the following that the cost γ_j of job J_j is a function of the (vector of) completion times C). However, as these users may compete for the same resources, it is generally not possible to simultaneously minimize the cost of each user. In a multi-user context, optimality is

[3] In [118], rather than considering the slowdown of a job J due to all the other jobs submitted to the system, the authors only consider the slowdown induced by the jobs released later than J. With this definition, a schedule whose maximum slowdown is λ is called λ-fair.

not defined as simply as in the single-user context, and it is common to use *Pareto-optimality*. In other words, C is Pareto optimal if it is impossible to strictly decrease the cost of a user without strictly increasing that of another. Any non-Pareto-optimal schedule can thus be considered as non-efficient as strictly a better usage of resources could be done.

In the case of max-based metrics, this means that the first maximum should be minimized, then the second should be minimized, and so on. Sum-based metrics obviously do not suffer from this flaw and always produce Pareto-optimal schedules. Therefore, rather than classical max-based metrics, one should rather consider their Pareto versions. These scheduling metrics are likely to be much more difficult (but also much more meaningful) than the classical ones as we do not have to only optimize the cost of the most constraining job but to optimize the cost of all jobs at the same time. (Table 4.2 in Section 4.10 compares the respective performance of a classical max-stretch minimization and of the corresponding Pareto optimization.)

Special-purpose metrics

Efficiency. Besides execution time and flow time metrics, many other types of objectives have been defined. It is well known that parallelism wastes resources. Indeed, except when it is taking advantage of such side effects as significant decrease of cache misses, a parallel execution is cumulatively using more resources than a sequential one (see for instance Amdhal's law). As a consequence, several objectives have been designed to measure whether schedules are resource efficient. Fundamentally, there exist many objectives around the efficiency notion because nobody seems to have been able to derive an efficiency-like variant meaningful even on unrelated machines. On our heterogeneous platforms, we are missing the good old ratio between the number of processors used and the speed-up that was used on parallel machines.

The *efficiency* defined in [106] is the ratio of the total utilization time of processors and of the makespan of the sequential execution on the fastest processor. To take processor heterogeneity into account, a processor utilization time is scaled by a factor proportional to its speed. Rather than considering wall clock times, the *total processor cycle consumption* [64] measures the total number of instructions processed by the different jobs on the different processors. Both approaches are doomed on unrelated machines such as processors with different instruction sets. Furthermore, in both cases nothing is said of communication resources: a communication-intensive parallel execution may appear efficient with those definitions even if most of the time its constituting jobs lay idle, the system waiting for their input data to be received before starting the jobs.

A seemingly closely related objective, *resource utilization*, is the fraction of the overall available resources that is effectively used. Following [63], in an open system, i.e., where jobs are arriving over time, the resource utilization is strongly correlated to the load submitted to the system, unless the system is saturated. A better measure would then be the *throughput*, that is, the number of jobs processed per unit of time. Given a platform, this also amounts to measure how efficiently the resources are used, here at the saturation point. When different types of jobs are submitted, or different users submit jobs, a simple maximization of throughput may lead to job starvation and this measure should be in some way counter-balanced by some fairness criteria, e.g., as in [11].

Lateness. In systems where jobs have due dates or deadlines, if the constraints on all jobs cannot be satisfied, the objective will be to maximize the number of satisfied constraints. The lateness of a job is the time elapsed between its due date and its completion time. One can then try to optimize the maximum lateness, the number of late jobs, the weighted number of late jobs (to take potential penalties into account), etc. [31].

Load balancing. A lot of algorithmic work focus on balancing the load among the processors involved in the parallel execution of an application. In order to measure how well a load is distributed, one can use a *load-balancing level* defined [107] as the ratio of the standard deviation of the processor loads and of the average load. In my point of view, load-balancing is the mean (for instance to achieve good makespans) and not an objective. To measure something as the load-balancing level may just help explain the performance reported for the true goal.

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 [128] or just to decrease the huge energy bills of supercomputers [43]. 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* [32]), or a posteriori to assess which scheduling policy is the most energy-efficient [43].

Fairness is rather a secondary objective, but may be an important one as classical objectives such as sum-based metrics can lead to starvation. Also, "fairness is important because it is inherent in the notion of sharing, which is the raison d'etre of the Grid" [53]. Rather than just considering the aggregate value of the objective function, one will look at how this objective compares among the different applications or the different users.

In many cases, one will not focus on a single objective, but will simultaneously consider several of them in a bi-criteria or multi-criteria approach: makespan and energy consumption, makespan and resource utilization, throughput and fairness, etc. Two approaches are then possible. A first one is to build a composite objective such as a linear combination of the different objectives; this has the disadvantage to look like mixing apples and bananas. The second approach consists in optimizing an objective while fixing bounds on the other objectives.

The above list of objective functions is not exhaustive. For peculiar systems or uses, specific objectives may be designed. For example, let us consider systems where data should be received at a given throughput, like in video streaming applications. Not only should the throughput be satisfied, but also the atomic data blocks should be received quite regularly to avoid any jagged motion perception. Then one may want to optimize the response time variability [49] which measures the local variations around the global throughput achieved.

1.4 Modeling issues

In this section I will highlight some of the main problems we may encounter while trying to model our scheduling problems.

Applications

In all models described in Section 1.1, there was the underlying assumption that we knew not only the *structure* of applications but that we also had some knowledge on the running time of its constituting tasks. Obviously there is a need to be able to predict the running time of applications if we are to use a model such as that of moldable tasks: we need to have some mean to decide how many processors to use, as using too many processors may slow down the application execution rather than speeding up things, and as, using more processors almost always decreases the efficiency of parallelization: there is then certainly a trade-off between the application execution time and the wastage of resources. The model is called *clairvoyant* if all characteristics are known and *non-clairvoyant* otherwise. As surprising at it may seem, even in the non-clairvoyant case it is sometimes possible to do clever scheduling as is illustrated by articles such as "scheduling in the dark" [56] and [118]. Most of the existing studies are obviously made in the clairvoyant framework. If the scheduler is not given any knowledge on task durations, it may try to guess these characteristics (e.g., see [83, 81] for references). Counter-intuitively, even when users give the scheduler knowledge on the duration of tasks, it may be better to use estimates of execution times given by very simple automatic predictors rather than user-provided estimates [131]. This is because user estimates are generally inaccurate [131].

Computing resources

If we do not want to go as far as modeling hierarchical memories or as modeling the time needed to access data on disks [100], the main issue we have with the modeling of computing resources is the instantiation of the parameters describing the performance of processors. This is rather easy for uniform machines, but can become intractable on unrelated machines.

The problem becomes even more complicated when the platform one deals with is a desktop Grid or a volunteer computing Grid. Such a Grid is composed of computing resources which can be reclaimed at any time by their owners. These resources are thus only temporarily available and their efficient use requires a modeling of their availability such as [135]. More generally, the larger the number of resources, the higher the probability of failures. Hence the need for some models of failure occurrence [59].

Besides the problem of execution time prediction, we may thus have to deal with dynamic, volatile, and failure-prone platforms.

Communication resources

As we have already hinted, we have to deal with several issues when modeling communications. Even if we have a single communication link in our platform, we have to choose a model predicting communication times, and then we have to instantiate this model. Both problems being non trivial. If the platform's interconnection network is not so simplistic, we have to use some graph-like representation of its structure. In most cases, however, the description of the structure and characteristics of the network interconnecting the different resources of a distributed platform is usually not available to users, mainly because of security reasons. We then have to guess the missing knowledge, but the problem of building a network representation which enables to make significantly accurate predictions is not yet solved [C7].

To make matters worse, communication links are seldom dedicated. For instance, if the first half of the computing resources inside a cluster are dedicated to a user, the communications induced by the computations happening in the second half can still interfere with communications in the first half. Therefore, in most cases the apparent characteristics of networks, in other words what is effectively available to applications, is time-evolving.

Moreover, the behavior of communication networks is technology-dependent, and technology is evolving. (For instance, bandwidths are dramatically increasing while latencies do not significantly decrease, hence the relative importance of latencies is increasing [38].) When we are addressing problems from a theoretical or fundamental point of view, we should then ask ourselves whether we should consider an idealized view of networks —i.e., considering what should or could happen in an idealized world— or a pragmatist down-to-earth approach —i.e., considering the sad reality of today's technique with all its flaws. Should we only focus on what can be commonly realized in practice in today's engineering world? Could we allow ourselves to wander in the less charted territories of what will (hopefully) be available in years to come?

Conclusion

The main issues we may have with models are thus the many sources of inaccuracies (task execution times, network characteristics, etc.), an only partial knowledge (structure of interconnection network, etc.), and the dynamicity of the platforms (network characteristics, machines failure, etc.). On the distributed heterogeneous platforms we target, defining accurate models of applications and platforms is a very difficult problem. Therefore, the actual applications and models may be quite different from the models used to represent them in many research work. There are two obvious reactions to this discrepancy. The first one is to claim that simple models, and even simplistic ones, are good enough and that one should not worry about more accurate ones. This claim is not generally true, as we will see on an example in Section 2.2. Therefore, we definitively need reasonably accurate models in order to reasonably predict what will happen during the execution of applications. The second reaction is to claim that if the platforms and applications are too complicated, if one cannot derive a neat theoretically guaranteed solution to our scheduling problem, then the only solution is to use some purely dynamic, system-like solution. However, on heterogeneous platforms purely dynamic solutions can be far off the best achievable performance, as we will see on an example in Section 3.3.

Whatever the difficulty of correctly modeling applications and platforms, I believe that we should still move towards using more accurate models. In the next section, I will try to list some of the possible research directions that we may follow to try to broaden the scope of the results established so far on scheduling for distributed heterogeneous platforms.

1.5 Uncertainties, dynamicity, and new research directions

Because of the different sources of uncertainties, scheduling algorithms are taking decisions based on faulty data. For instance, algorithms can have misperceptions on what are the relative execution times of jobs, or on which computer a job will run the fastest. This may cause the algorithms not to consider the jobs in the correct order, not to map them on the right processors, or not to have a good evaluation of their weight in the objective function (for objective functions like the stretch). One can wish to study how often such misperceptions happen, as in [83]. What is more important is the potential consequences of these misperceptions, and more generally of the uncertainties, on the performance of the scheduling algorithms.

From what was written so far, the reader may have the feeling that most of the existing scheduling solutions are meaningless because they were derived using incomplete models and while overlooking the imperfections of these models. Such a conclusion would be hasty for two very different reasons. Firstly, there are practical situations where application and platform characteristics can be known and modeled accurately (for instance Section 3.3.6), or where inaccuracies and uncertainties do not really matter (for instance when you just want the computation to be performed but have no, or very loose, requirements on execution times). Secondly, if there is no a priori guarantee that any solution derived with a faulty model will have a very good performance in an actual setting, conversely, there is no a priori guarantee that it will only achieve a very poor performance.

To move towards more accurate models and more realistic solutions, for the most difficult scheduling problems, we need to be able to handle, or at least to cope with, many sources of uncertainties and of dynamicity. The main alternative we have is to either try to prevent a priori the consequences of these uncertainties and dynamicity, or only to correct a posteriori these consequences.

A first approach would be to assess whether the solutions derived so far may or no suffer from the discrepancies between models and reality. This is the aim of a *sensitivity analysis* which will study how the performance of an algorithm evolves when the actual characteristics of the objects it deals with varies from the model used. A more active approach would be to purposely build *robust* algorithms, that is, algorithms that can cope with these uncertainties. In the so-called *robust optimization* framework [84, 137], the uncertainty is modeled via a set of scenarios. A solution to an optimization model is then defined as *solution-robust* if it remains "close" to optimal for all scenarios of the input data, and as *model-robust* if it remains "almost" feasible for all data scenarios. Another, more radical, approach to the design of robust (scheduling) algorithms, can be exemplified by the "Internet-based Computing" series of work by Arnold Rosenberg, Grzegorz Malewicz, et al. (for instance, see [98, 47]). In this work, the authors schedule a task graph while assuming that they have no knowledge on task execution times. They schedule the task graph such that, after any scheduling decision, as many tasks as possible are available for scheduling. This way they hope that, whatever the task actual execution times, they will always have tasks ready to be executed, each time new computing resources become available. Fundamentally, they try to counterbalance uncertainties on task execution times by giving the maximum possible freedom to the scheduler. As in other scheduling works targeting systems with only very partial knowledge, such as [56, 113], the underlying idea is to minimize the risk taken while making any scheduling decision.

When considering the problems of uncertainties, dynamicity, volatility, and failures, an obvious move would be to go towards stochastic models. In other words, rather than using fix values one would use distributions to describe characteristics such as task execution time or link bandwidth. Such an approach is very appealing as it simultaneously gives tool to model uncertainties and dynamicity, and thus to address most, if not all, of our problems. However, we are just shifting the problems. Indeed, moving to stochastic representations of application and platform characteristics create two new and difficult problems: 1) what are the relevant stochastic models ? 2) how do we handle them to solve our scheduling problems ?

Whatever the characteristics we would like to stochastically model, there are a wide variety of distribution shapes we can choose from. The characteristics we would like to model are very diverse. Indeed, to move from static application models to stochastic ones, we could need to have models of things such as the distribution of task execution times, of errors in user predictions, or of task failures. To move from static platform models to stochastic ones, one could need to model the evolution over time of network characteristics, of the impact of processor hardware on task execution times, or of the time a computer remains available in a desktop Grid. There are no a priori reasons for so different phenomena to be

perfectly described by the very same probability law. Furthermore, we can ask ourselves what we can hope to gain by replacing an inaccurate and static representation by a stochastic model whose only justification is that "this is the one everybody uses" (justification actually heard in this very context). Therefore, a priori, we should carefully select the right distribution for each of the modeled characteristics. This should be done while keeping in mind that there may be characteristics for which choosing one probability distribution rather than another one will have a significant impact on the actual performance of the solution designed, while for other characteristics this may not be the case. From this discussion, it sounds that a lot of our problem characteristics should be carefully studied and modeled. Several of the necessary studies have already be done or at least attempted, such as the modeling of jobs inter-arrival times on a Grid [95] (and not just on a cluster or a single parallel computer), or the prediction of the running time of applications, even on distributed and heterogeneous platforms (see for instance [51] and the reference therein). The set of available results, however, is far from being comprehensive, and much remains to be done. See Dror Feitelson's book [59] for a survey of existing studies.

The second problem we have with the stochastic approach, is the study of the stochastically defined scheduling problems. Scheduling problems were often already hard on parallel computers. They were obviously significantly harder on statically defined distributed and heterogeneous platforms. One can bet that they won't become simpler when the problem characteristics will only be known through some probability laws.

Let us specifically focus, for a moment, on the work of the GRAAL team, in which was done the work exposed in the remaining chapters of this thesis. One of the trademarks of this work is the recurring use of linear programming. At first sight, it may not seem obvious to mix linear programming and stochastic models. However, this is what chance-constrained programming [75] does. This is an approach to deal with random parameters in optimization problems. One of the difficulties with stochastic models is that decisions must be taken prior to the observation of the random parameters. Therefore, one can hardly find any decision which would definitely exclude later constraint violations caused by unexpected random effects. As we can no longer insure that all constraints will always be met, we will relax our system and only target solutions which guarantee that constraints are satisfied with high probability, i.e., that only a low percentage of realizations of the random parameters leads to constraint violations under this fixed decision. In the general case, chance-constrained programming is a very difficult problem. Exact resolution methods are only known for very special cases or distribution functions. It would nevertheless be very interesting to see whether and how chance-constrained programming can allow us to extend some of our previous works.

More generally, it would be worth investigating whether stochastic programming —which is defined by [108] as "a framework for modelling optimization problems that involve uncertainty"— can give us some tools and methods to tackle some stochastic scheduling problems.

Several existing scheduling studies address stochastic models. In the field of distributed heterogeneous platforms, these studies are still quite rare. The existing work related to Grid computing uses stochastic approaches, for instance, to describe job arrival times [8, 74], task execution times [119], job waiting times [67], processor computing power [115], processor failures [52], or to describe complex application models [116]. Most of the problems, however, remain to be investigated in the context of stochastically described platforms and applications.

Stochastic approaches are not, by far, the only means to address uncertainties and dynamicity. There already exist a lot of work in scheduling considering dynamic systems and working outside the scope of stochastic models. Most of this work answers to the dynamicity of applications and/or platforms by using dynamic schedulers (sometimes also called adaptive schedulers). The fact that many studies consider dynamic schedulers (see for instance [132, 18, 77, 39, 79]) can be explained either because the targeted systems are fundamentally dynamic or because designing dynamic solutions is far more easy. Under the dynamic banner, we find all the approaches which try to correct a posteriori all the consequences of dynamicity and uncertainties, such as load-balancing algorithms (for instance, see the references in [30]). The range of dynamic solutions is quite diverse. Some works will consider that schedulers have feedback from the system, giving them information about the current state of a computation, while other work in the same context will consider that the scheduler is left in the dark (see for instance the works [1] and [113], which both target the dynamic and time-evolving allocation of processor to parallel tasks). Dynamic schedulers, that is, schedulers which take decisions at runtime while considering the state of

the system, can be divided into two categories: those which adapt a previously and statically defined solution, and the purely dynamic ones. The fundamental problem with purely dynamic approaches on heterogeneous platforms, as we have already stated, is that they can be far of the best achievable performance (as we will see on an example in Section 3.3). This is because these approaches can not forecast the pitfalls laid by the heterogeneity of platform and/or application (cf. Section 3.3), or cannot discover at runtime, through trial and errors, the sophisticated solutions which are the only ones able to truly harness the power of the computing platforms. Because of these inherent limitations of purely dynamic solutions, although platforms and applications are (in part) dynamic in nature, and although they are only partially known, our credo is that static (or mixed-static and dynamic) approaches are still better than purely dynamic ones.

Among the other means, tools, and methods one can try to use to cope with uncertainties, there is the game theory approach. This approach is mainly used in two different contexts: to build decentralized schedulers and to try to diminish uncertainties by inciting users to report the true characteristics of their jobs. Some of the existing work deal with task scheduling on (identical) parallel machines [3, 35], divisible load scheduling on heterogeneous processors [34], or the study of the properties of decentralized scheduling [114]. The interested reader can see [76] for an introduction to the use of game theory in scheduling. Some studies also consider market-like approaches to incite users to report accurate characteristics [136, 70]. And I will not write about meta-heuristics and other genetic-algorithm approaches such as [138].

1.6 Conclusion

It should now be obvious to the reader that the problem of scheduling scientific computations on distributed heterogeneous platforms is far from being solved.

In the remaining of this thesis, I will present some of the results I achieved for this problem. The following chapters do not cover all the research work I have ever done, not even all the work I did since I defended my PhD. As I decided to focus in this thesis on what had been my main research subject the last five years, this thesis is not even a best-of: I have left aside some work I am very proud of, like the work Martin Rinard and I did on the pointer and escape analysis of Java programs [C22]. I even left aside work perfectly falling under the scope of this thesis, either because I did not feel it was illustrative enough (like the beautiful algorithm Loris Marchal, Veronika Rehn, Yves Robert, and myself designed to redistribute identical tasks on master-slave platforms [B5]) or because is was not deemed mature enough (like the work Matthieu Gallet, Loris Marchal, and myself, are doing on the use of task duplication to speed up computations).

I have decided to present here three series of work. In the first one (Chapter 2), I present two studies which respectively show the impact application models and platform models can have on our ability to efficiently solve scheduling problems. The second series of work (Chapter 3) is made of three independent studies around the problem of taking into account the communications induced by application input data. An idea underlying this chapter is that, in distributed platforms, computation scheduling cannot usually be separated from data management and communication scheduling. The last chapter (Chapter 4) is devoted to a comprehensive study of the online scheduling of divisible loads, the aim being to minimize the stretch of jobs. Therefore, in this chapter we simultaneously deal with the dynamicity due to arriving-over-time applications and with a not so classical objective function.

Although the problem instances that were so far targeted in the literature were already quite difficult, I believe that we should still move towards using more accurate application and platform models, and thus towards more difficult scheduling problems, in order to try to broaden the range of actual scheduling problems that we are able to efficiently address.

Chapter 2

The impact of models

To be able to design efficient, and sometimes sophisticated, scheduling strategies, we must have information on the applications to be scheduled and on the platform used to process them. We are not directly manipulating applications and platforms, we are obviously using models. In this chapter, we will see that the models used to represent either applications or platforms are very important, as the choice of a model can greatly impact both the algorithmics and the performance obtained.

In Section 2.1, we consider a simple master-worker scheduling problem. We study this problem first under the classical scheduling framework, then under the divisible load theory framework. The two frameworks correspond to two different models of the targeted application: either data and their processings are atomic, or they can be arbitrarily divided. Depending on the model used, we are or not able to solve our optimization problem in all its generality, and we derive more or less efficient algorithms.

In Section 2.2, we study how the complexity of a load-balancing problem evolves when the interconnection network model becomes more accurate and thus more complex. As our problem very soon becomes NP-complete, one may wonder whether using an accurate model —which forbids us to derive "optimal" solutions in polynomial-time— does not make us lose more than we can gain. Through the evaluation of the heuristics proposed for the different network models, we show that, in our context, we must use accurate platform models in order to design better heuristics and to better harness the processing power of platforms.

Both studies show that one must carefully choose the application and platform models one works with: models must be realistic enough for the derived results to be meaningful, but they should be amenable to analysis so as to enable the design of efficient solutions.

2.1 An example of the impact of the application model: classical scheduling vs. divisible load

Bibliographical note: The missing details and proofs can be found in [B12, D20].

2.1.1 Introduction

Among the usual operations found in parallel codes is the *scatter* operation, which is one of the *collective* operations usually shipped with message passing libraries. For instance, the mostly used message passing library MPI [102] provides an `MPI_Scatter` primitive that allows the programmer to distribute even parts of data to the processors in the MPI communicator. The typical usage of the scatter operation is to spawn an SPMD computation section on the processors after they received their piece of data. Thereby, if the computation load on processors depends on the data received, the scatter operation may be used as a means to load-balance computations, provided the items in the data set to scatter are independent. MPI provides the primitive `MPI_Scatterv` that allows to distribute *unequal* shares of data. Starting from an existing application, our problem is thus to load-balance the execution by computing a data distribution depending on processor speeds and network link bandwidths. This study gives us the opportunity to compare two application models: the classical scheduling model where data and computations are supposed to be atomic, and the divisible load theory.

In Section 2.1.2 we present our target application, a real scientific application in geophysics, written in MPI, that we ran to ray-trace the full set of seismic events of year 1999. We study our problem using the classical scheduling approach in Section 2.1.3, and the divisible load theory in Section 2.1.4. We then present experimental results in Section 2.1.5 before concluding in Section 2.1.6.

2.1.2 Motivating example

Seismic tomography

The geophysical code we consider is in the seismic tomography field. The general objective of such an application is to build a global seismic velocity model of the Earth interior. The various velocities found at the different points discretized by the model (generally a mesh) reflect the physical rock properties in those locations. The seismic waves' velocities are computed from the seismograms recorded by captors located all around the globe: once analyzed, the wave type, the earthquake hypocenter, and the captor locations, as well as the wave travel time, are determined. From these data, a tomography application reconstructs the event using an initial velocity model. The wave propagation from the source hypocenter to a given captor defines a path, that the application evaluates given properties of the initial velocity model. The time for the wave to propagate along this evaluated path is then compared to the actual travel time and, in a final step, a new velocity model that minimizes those differences is computed. This process is more accurate if the new model better fits numerous such paths in many locations inside the Earth, and is therefore very computationally demanding.

The example application

We now outline how the application under study exploits the potential parallelism of the computations, and how the tasks are distributed across processors. Recall that the input data is a set of seismic waves' characteristics each described by a pair of 3D coordinates (the coordinates of the earthquake source and those of the receiving captor) plus the wave type. With these characteristics, a seismic wave can be modeled by a set of *ray paths* that represents the wavefront propagation. Seismic waves' characteristics are sufficient to perform the ray-tracing of the whole associated ray path. Therefore, all ray paths can be traced independently. The existing parallelization of the application (presented in [69]) assumes an homogeneous set of processors (the implicit target being a parallel computer). There is one MPI process per processor. The following pseudo-code outlines the main communication and computation phases:

```
if (rank = MASTER)
  raydata ← read n lines from data file;
MPI_Scatter(raydata,n/p,...,rbuff,...,MASTER,MPI_COMM_WORLD);
compute_work(rbuff);
```

where p is the number of processors involved, and n the number of data items. The `MPI_Scatter` instruction is executed by the master and the computation processors, and structure the set of processors as a master-worker system. The processor identified as `MASTER` performs a send of contiguous blocks of $\lfloor n/p \rfloor$ elements from the `raydata` buffer to all workers while all workers make a receive operation of their respective data in the `rbuff` buffer. For sake of simplicity the remaining ($n \bmod p$) items distribution is not shown here. Figure 2.1 shows a potential execution of this communication operation, with P_4 as master.

Communication model

Figure 2.1 outlines the behavior of the scatter operation as it was observed during the application runs on our test grid. In the MPICH-G2 implementation we used (v1.2.2.3), when it performs a scatter operation, the master (P_4 on the figure) must have completely sent one message before it can start sending another message. This behavior corresponds to the *one-port* model [21, 22]. As the master sends data to workers in turn, a worker actually begins its data reception after all previous workers have been served. This leads to a "stair effect" represented on Figure 2.1 by the end times of the receive operations (black boxes). The implementation makes the order of the destination workers follow the worker ranks. Also, the master can only start processing its share of data when it has completed the last of the communications in the scatter.

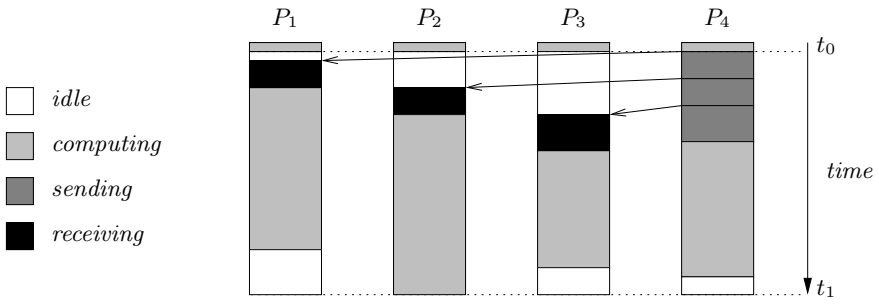


Figure 2.1: A scatter communication followed by a computation phase.

2.1.3 Study under the classical scheduling framework

After briefly presenting our framework, we give two dynamic programming algorithms, the second one being more efficient than the first one, but under some additional hypotheses on the cost functions.

Framework

In this paragraph, we introduce some notations, as well as the cost model used to further derive the optimal data distribution.

We consider a set of p processors: $P_1, \ldots, P_p$. Processor P_i is characterized by 1) the time $T_{\text{comp}}(i, x)$ it takes to compute x data items; 2) the time $T_{\text{comm}}(i, x)$ it takes to receive x data items from the master. We want to process n data items. Thus, we look for a distribution $n_1, \ldots, n_p$ of these data over the p processors that minimizes the overall computation time. All along this section the master will be the last processor, P_p (this simplifies expressions as P_p can only start to process its share of the data items *after* it has sent the other data items to the workers). As the master sends data to workers in turn, worker P_i begins its communication after workers $P_1, \ldots, P_{i-1}$ have been served, which takes a time $\sum_{j=1}^{i-1} T_{\text{comm}}(j, n_j)$. Then the master takes a time $T_{\text{comm}}(i, n_i)$ to send to P_i its data. Finally P_i takes a time $T_{\text{comp}}(i, n_i)$ to process its share of the data. Thus, P_i ends its processing at time:

$$T_i = \sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i). \tag{2.1}$$

The time, T, taken by our system to compute the set of n data items is therefore:

$$T = \max_{1 \leq i \leq p} T_i = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i) \right), \tag{2.2}$$

and we are looking for the distribution $n_1, \ldots, n_p$ minimizing this duration.

An exact solution by dynamic programming: basic algorithm

Equation (2.2) can be rewritten:

$$T = T_{\text{comm}}(1, n_1) + \max \left(T_{\text{comp}}(1, n_1), \max_{2 \leq i \leq p} \left(\sum_{j=2}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i) \right) \right).$$

So, the time to process n data on processors 1 to p is equal to the time taken by the master to send n_1 data items to P_1 plus the maximum of 1) the time taken by P_1 to process its n_1 data items; 2) the time for processors 2 to p to process $p - n_1$ data. This leads to the dynamic programming Algorithm 1 (the distribution is expressed as a list, hence the use of the list constructor "cons"). In Algorithm 1, $cost[d, i]$ denotes the cost of the processing of d data items over the processors P_i through P_p. $solution[d, i]$ is a list describing a distribution of d data items over the processors P_i through P_p which achieves the minimal execution time $cost[d, i]$.

Algorithm 1: Compute an optimal distribution of n data over p processors.

```
1  solution[0, p] ← cons(0, NIL)
2  cost[0, p] ← 0
3  for d ← 1 to n do
4  |  solution[d, p] ← cons(d, NIL)
5  |  cost[d, p] ← T_comm(p, d) + T_comp(p, d)
6  for i ← p − 1 downto 1 do
7  |  solution[0, i] ← cons(0, solution[0, i + 1])
8  |  cost[0, i] ← 0
9  |  for d ← 1 to n do
10 |  |  (sol, min) ← (0, cost[d, i + 1])
11 |  |  for e ← 1 to d do
12 |  |  |  m ← T_comm(i, e) + max(T_comp(i, e), cost[d − e, i + 1])
13 |  |  |  if m < min then
14 |  |  |  |  (sol, min) ← (e, m)
15 |  |  solution[d, i] ← cons(sol, solution[d − sol, i + 1])
16 |  |  cost[d, i] ← min
17 return (solution[n, 1], cost[n, 1])
```

Algorithm 1 has a complexity of $O(p \cdot n^2)$, which may be prohibitive. But Algorithm 1 only assumes that the functions $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are nonnegative and null whenever $x = 0$.

An exact solution by dynamic programming: optimized algorithm

If we make the assumption that $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are non-decreasing with x —which seems quite reasonable— we can make some optimizations on the algorithm. These optimizations consist in reducing the bounds of the inner loop (e-loop, lines 11–13 of Algorithm 1). Algorithm 2, on page 23, presents these optimizations.

In the worst case, the complexity of Algorithm 2 is the same than for Algorithm 1, i.e., $O(p \cdot n^2)$. In the best case, it is only $\Theta(p \cdot n)$. We implemented both algorithms, and in practice Algorithm 2 is far more efficient (see Section 2.1.5).

2.1.4 Study under the divisible load model

In the previous section we studied our problem under the classical scheduling approach. Under this approach we were able to design a polynomial-time dynamic programming algorithm which derives an optimal schedule *when* one has fixed the order under which the workers should receive the data items. We now study our problem in the framework of the divisible load theory. In this context, we will see that the processor ordering has an impact on the overall performance and we will determine the optimal ordering.

Here, we solve our problem in the *divisible load* framework. Therefore, we make the hypothesis that all the functions $T_{\text{comm}}(i, n)$ and $T_{\text{comp}}(i, n)$ are linear in n. In other words, we assume that there are constants c_i and w_i such that $T_{\text{comm}}(i, n) = c_i \cdot n$ and $T_{\text{comp}}(i, n) = w_i \cdot n$. (Note that $c_p = 0$ with our hypotheses.) Also, we only look for a rational solution and not an integer one as we previously did. In other words, in the context of classical scheduling we assumed that the data items were atomic, which they are. Under the divisible load theory, we assume that we can arbitrarily divide the overall amount of data: we allow the study to tell us to only allocate to a processor a fraction of a ray, even if this is meaningless.

With our hypotheses, we are able to establish some properties of optimal solutions. It is often stated in papers dealing with the divisible load theory that in an optimal solution all processors work and end their computations simultaneously. So we first consider this particular case.

Theorem 1 (Execution duration)**.** *If we are looking for a rational solution, if each processor P_i receives a (non empty) share n_i of the whole set of n data items and if all processors end their computation at a*

Algorithm 2: Compute an optimal distribution of n data over p processors (optimized version).

```
1  solution[0,p] ← cons(0, NIL)
2  cost[0,p] ← 0
3  for d ← 1 to n do
4  |  solution[d,p] ← cons(d, NIL)
5  |  cost[d,p] ← T_comm(p,d) + T_comp(p,d)
6  for i ← p − 1 downto 1 do
7  |  solution[0,i] ← cons(0, solution[0,i+1])
8  |  cost[0,i] ← 0
9  |  for d ← 1 to n do
10 |  |  if T_comp(i,0) ≥ cost[d,i+1] then
11 |  |  |  (sol,min) ← (0, T_comm(i,0) + T_comp(i,0))
12 |  |  else if T_comp(i,d) < cost[0,i+1] then
13 |  |  |  (sol,min) ← (d, T_comm(i,d) + cost[0,i+1]))
14 |  |  |  else
15 |  |  |  |  (e_min,e_max) ← (0,d)
16 |  |  |  |  e ← ⌊d/2⌋
17 |  |  |  |  while e ≠ e_min do
18 |  |  |  |  |  if T_comp(i,e) < cost[d−e,i+1] then
19 |  |  |  |  |  |  e_min ← e
20 |  |  |  |  |  else
21 |  |  |  |  |  |  e_max ← e
22 |  |  |  |  |  e ← ⌊(e_min + e_max)/2⌋
23 |  |  |  |  (sol,min) ← (e_max, T_comm(i,e_max) + T_comp(i,e_max))
24 |  |  for e ← sol − 1 downto 0 do
25 |  |  |  m ← T_comm(i,e) + cost[d−e,i+1]
26 |  |  |  if m < min then
27 |  |  |  |  (sol,min) ← (e,m)
28 |  |  |  else if cost[d−e,i+1] ≥ min then
29 |  |  |  |  break
30 |  |  solution[d,i] ← cons(sol, solution[d−sol,i+1])
31 |  |  cost[d,i] ← min
32 return (solution[n,1], cost[n,1])
```

same date t, then the execution duration is

$$t = \frac{n}{\sum_{i=1}^{p} \frac{1}{c_i+w_i} \cdot \prod_{j=1}^{i-1} \frac{w_j}{c_j+w_j}} \tag{2.3}$$

and processor P_i receives

$$n_i = \frac{1}{c_i+w_i} \cdot \left(\prod_{j=1}^{i-1} \frac{w_j}{c_j+w_j} \right) \cdot t \tag{2.4}$$

data to process.

In the rest of this section we note:

$$D(P_1, \ldots, P_p) = \frac{1}{\sum_{i=1}^{p} \frac{1}{c_i+w_i} \cdot \prod_{j=1}^{i-1} \frac{w_j}{c_j+w_j}}.$$

and so we have $t = n \cdot D(P_1, \ldots, P_p)$ under the hypotheses of Theorem 1.

The question, answered by the next theorem, is when Theorem 1 can be used to find a rational solution to our system.

Theorem 2 (Simultaneous endings)**.** *Given p processors, $P_1, \ldots, P_i, \ldots, P_p$, whose communication and computation duration functions $T_{\text{comm}}(i,n)$ and $T_{\text{comp}}(i,n)$ are linear in n, there exists an optimal rational solution where each processor receives a non-empty share of the whole set of data, and all processors end their computation at the same date, if and only if*

$$\forall i \in [1, p-1], \quad c_i \leq D(P_{i+1}, \ldots, P_p).$$

The proof of Theorem 2 [B12, D20] shows that any worker P_i satisfying the condition $c_i > D(P_{i+1}, \ldots, P_p)$ is not interesting for our problem: using it will only increase the whole processing time. The underlying intuition is obvious. Indeed, if $c_i > D(P_{i+1}, \ldots, P_p)$ then it is more time-expensive to send some data to worker P_i than to have these same data being sent and processed by the set of the last processors $P_{i+1}, ..., P_p$. Therefore, we just forget those uninteresting processors and Theorem 2 states that there is an optimal rational solution where the remaining processors are all working and have the same end date.

So far, once we have a processor ordering, we have closed-form formulas to tell us which processors participate in an optimal solution and what amount of data they should each process. We still have to study the impact of the processor ordering: Equation (2.3) shows that in our case the overall computation time is not symmetric in the processors but depends on their ordering. The optimal ordering is dictated by the sole communication capabilities (as this is the case in other scheduling problems on heterogeneous networks [6, 78]).

Theorem 3 (Processor ordering policy)**.** *When all functions $T_{\text{comm}}(i,n)$ and $T_{\text{comp}}(i,n)$ are linear in n, and when we are only looking for a rational solution, then the smallest execution time is achieved when the workers (not considering the master) are ordered in non-increasing order of their bandwidths (from P_1, one of the workers connected to the master with the highest bandwidth, to P_{p-1}, one of the workers connected to the master with the smallest bandwidth), the last processor being the master. The respective processor processing powers are not taken into account.*

We will comment on the results of this section in the conclusion (Section 2.1.6).

2.1.5 Experimental results

Hardware environment

Our experiment consists in the computation of 817,101 ray paths (the full set of seismic events of year 1999) on 16 processors. All machines run Globus [62] and we use MPICH-G2 [86] as message passing library. Table 2.1 shows the resources used in the experiment. The input data set is located on the PC named *dinadan*, at the end of the list, which serves as master. The computers are located at two geographically distant sites. Processors 1, 2, 3, and 16 (standard PCs with Intel PIII and AMD Athlon XP), and 4 and 5 (two Mips processors of an SGI Origin 2000) are in the same premises, whereas processors 6 to 13 are taken from an SGI Origin 3800 (Mips processors) named *leda*, at the other end of France. Processor and network link performance are values computed from a series of benchmarks we performed on our application. Notice that *merlin*, with processors 14 and 15, though geographically close to the master, has the smallest bandwidth because it happened to be connected to a 10 Mbit/s hub during the experiment whereas all others were connected to fast-Ethernet switches.

The column w indicates the number of seconds needed to compute one ray (the lower, the better). The associated rating is simply a more intuitive indication of the processor speed (the higher, the better): it is

Table 2.1: Processors used as computational nodes in the experiment.

Machine	CPU #	Type	w (s/ray)	Rating	c (s/ray)
caseb	1	XP1800	0.004,629	2.00	$1.00 \cdot 10^{-5}$
pellinore	2	PIII/800	0.009,365	0.99	$1.12 \cdot 10^{-5}$
sekhmet	3	XP1800	0.004,885	1.90	$1.70 \cdot 10^{-5}$
seven	4, 5	R12K/300	0.016,156	0.57	$2.10 \cdot 10^{-5}$
leda	6–13	R14K/500	0.009,677	0.95	$3.53 \cdot 10^{-5}$
merlin	14, 15	XP2000	0.003,976	2.33	$8.15 \cdot 10^{-5}$
dinadan	16	PIII/933	0.009,288	1.00	0.00

the inverse of w normalized with respect to a rating of 1 arbitrarily chosen for the Pentium III/933. When several identical processors are present on a same computer (6–13 and 14, 15) the average performance is reported.

The network link throughputs between the master and the workers are reported in column c assuming a linear communication cost. It indicates the time in seconds needed to receive one data element from the master. Considering linear communication costs is sufficiently accurate in our case since the network latency is negligible compared to the sending time of the data blocks.

Results

We do not report here on the experiment we made to show that taking the platform heterogeneity into account was cutting roughly by half the overall execution time [B12]. We will only discuss the two different types of solutions (classical scheduling vs. divisible load approach) and the impact of the processor ordering.

In order to compare the solutions of the two different approaches, we made the assumption that the computation and communication cost functions were affine and non-decreasing. With our large number of rays, on a Pentium III at 933 MHz, Algorithm 1 had a running time of more than two days (we interrupted it before its completion) and Algorithm 2 had a running time of 6 minutes, whereas computing the value defined by Equation 2.4 for each processor is instantaneous[1]. We then had to round to integers our rational values. This process leads in our case to an error, relative to the optimal solution, of less than $6 \cdot 10^{-6}$.

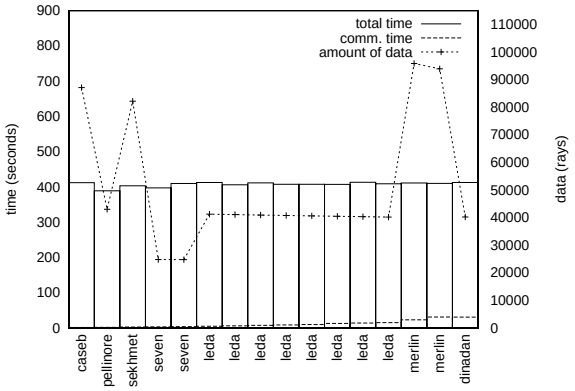


Figure 2.2: Load-balanced execution with nodes sorted by descending bandwidth.

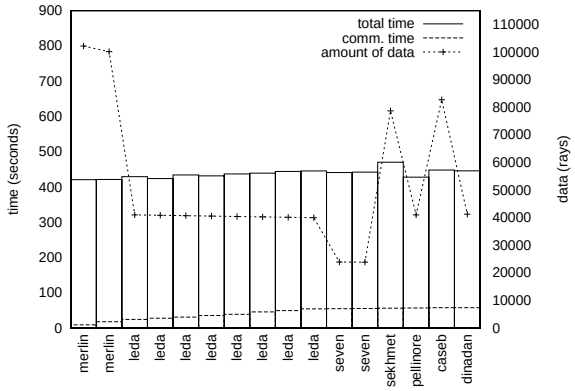


Figure 2.3: Load-balanced execution with nodes sorted by ascending bandwidth.

The master sends data to processors in turn and a worker actually begins its communication after all previous workers have received their shares of data. In the MPICH implementation of MPI, the order of the destination processors in scatter operations follows the processor ranks defined by the program(mer). Therefore, we can actually control the order in which the master serves the workers.

The experiments of Figures 2.2 and 2.3 evaluate the execution performance of the theoretically optimal processor ordering policy (the processors are ordered in non-increasing order of their bandwidths on Figure 2.2) by comparing this policy to the opposite one (the processors are ordered in non-decreasing order of their bandwidths on Figure 2.3). Note that whatever the order chosen, the master is always set at the end of the list (despite its infinite bandwidth to itself) since the scatter semantics forces this processor to wait until the completion of all communications to start processing its share of the data (see Section 2.1.2).

For the theoretically optimal processor ordering policy, the execution appears well balanced: the earliest and latest finish times are 388 s and 412 s respectively, which represents a maximum difference in finish times of 6% of the total duration. For the reverse policy, the load balance is acceptable with a maximum difference in ending times of about 10% of the total duration (the earliest and latest processors

[1] Actually, we did not bother to implement this computation. We rather solved in rational the linear program coding our problem, using `pipMP` [58, 109]. Obviously, these are two means to obtain the exact same result. Solving our linear program was instantaneous.

finish after 420 s and 469 s). As predicted, the total duration is longer (by 57 s) than with the processors in the reverse order. Though the load was slightly less balanced than in the previous experiment (because of a peak load on *sekhmet* during the experiment[2]), most of the difference comes from the idle time spent by processors waiting before the actual communication begins. This clearly appears on Figure 2.3: the surface of the bottom area delimited by the dashed line (the "stair effect") is larger than in Figure 2.2.

Predicted times

We now have a quick look at the differences between the times predicted by our model and the completion times achieved in reality.

For the communication times, the predictions were more accurate for the processors that were geographically close to the master with an error that was less than 0.7 s, while it was up to 5.5 s for the processors located at the other end of the country. This certainly comes from the fact that network performance are more stable (and thus more easily predictable) for a local network than for a wide area network. Compared to the small communication times we had, these errors look rather big. We can however see, from Figures 2.2 and 2.3, that, despite the possible mispredictions, it is important to take the communications into account to choose a good processor ordering and to compute a load-balanced data distribution.

For the computation times, the relative error was less than 2.3% on the Origins (*leda* and *seven*) while it was sometimes as great as 9.2% on the PCs. The difference here is due to the fact that the processors on the PCs were possibly shared with other users while a reservation system (LSF in this case) gave us a dedicated use of the processors on the parallel machines.

Concerning the whole execution time of the experiments, the prediction was pretty good for the first experiment: it was underestimated by less than 1.9%. As already seen for the load-balancing, some interferences took place during the second experiment, leading here to an underestimation of about 11.6%. With a static approach like the one presented in this section, the quality of the prediction clearly depends on the predictability of the network link bandwidths and the processor performance. These experiments show however that a good load-balance along with a close prediction of the end time are achievable in practice. Furthermore, these experiments validate our use of the divisible load theory to model our application.

2.1.6 Conclusion

In this section we studied a load-balancing problem using two different approaches: the classical scheduling framework and the divisible load theory. The main advantage of the classical scheduling framework is its accuracy: it does not make the approximation of considering that the data items can be arbitrarily divided. But its accuracy does not pay off. Indeed, under this classical approach we were not able to derive the optimal processor ordering, and thus we only partially solved our problem. Furthermore, the algorithms obtained are very computationally expensive when the solution obtained using the divisible load theory comes at almost no cost, and is of the very same quality for our application. For such an application, in such a context, there is no reasons to use the classical scheduling framework rather than the divisible load theory.

Finally, let us make some remarks on the results established under the divisible load model. In this model, *everybody knows* that in an optimal solution all processor works and complete their shares simultaneously. This is such an obvious result that authors rarely bother to prove it and, when they try, the proofs and/or arguments are quite often wrong (see for example [71, 88, 27] and our comments in [B12]). We have shown that in our context all *participating* processors complete their share of data simultaneously in an optimal solution but that in an optimal solution not all processors always participate.

[2]The reader may wonder why we did not just try to redo the experiment to have a cleaner and more convincing graph to present. There are basically two reasons. First, not any single experiment was successful as we were using not only desktop computers but also two clusters with batch scheduling systems: our jobs had to have access to nodes on the two clusters almost simultaneously for our experiment to be able to run. Quite often the processes on one cluster were killed by a time-out before processes on the other cluster were allocated processors and could start. The second reason is that one week after we finally obtained the graph of Figure 2.3, some parts of the network connecting French universities, Renater, was upgraded. The network was then so fast that the communication times did not matter any more for our problem. The gain due to the optimal processor ordering would have been hidden under the uncertainties on the experimental measures... (In the presented experiments the impact of the communication times were already far from being tremendous.)

And then some still wonder why we are not so enthusiastic about doing real-life experiments !

This peculiar result is due to our hypothesis, due to the version of MPI we used at the time, that the master processor can not communicate and compute simultaneously. If we remove this hypothesis, we fall under the framework studied by Beaumont, Legrand, and Robert in [14, 12]. There results are far more general than ours as they also studied the problem of the distribution of the divisible load in multi rounds.

2.2 An example of the impact of the platform model: load-balancing iterative applications

Bibliographical note: The missing details and proofs can be found in [112, D21, D19, B10].

In this section we investigate the mapping of iterative algorithms onto heterogeneous sets of computational resources. Such algorithms typically operate on a large collection of application data, which will be partitioned over the processors. At each iteration, some independent calculations will be carried out in parallel, and then some communications will take place. This scheme is very general, and encompasses a broad spectrum of scientific computations, from mesh based solvers (e.g., elliptic PDE solvers) to signal processing (e.g., recursive convolution), and image processing algorithms (e.g., mask-based algorithms such as thinning). Our aim here is twofold. The first one is to show the impact of the network model used on the problem complexity. The second one is to stress the importance of an accurate model.

An abstract view of our problem is the following: an iterative algorithm repeatedly operates on a large rectangular matrix of data samples. This data matrix is split into vertical slices that are allocated to the computing resources (processors). At each step of the algorithm, the slices are updated locally, and then boundary information is exchanged between consecutive slices. This (virtual) geometrical constraint advocates that processors be organized as a virtual ring. Then each processor will only communicate twice, once with its (virtual) predecessor in the ring, and once with its (virtual) successor. Note that there is no reason *a priori* to restrict to a uni-dimensional partitioning of the data, and to map it onto a uni-dimensional ring of processors: more general data partitionings, such as two-dimensional, recursive, or even arbitrary slicings into rectangles, could be considered. But uni-dimensional partitionings are very natural for most applications, and, as will be shown, the problem to find the optimal one is already difficult.

The target architecture is a fully heterogeneous platform, composed of different-speed processors that communicate through links of potentially different bandwidths. On the architecture side, the problem is twofold: (i) select the processors that will participate in the solution and decide for their ordering that will represent the arrangement into a ring; (ii) assign communication routes from each participating processor to its successor in the ring. One major difficulty of this ring embedding process is that some of the communication routes may have to share some physical communication links: indeed, the communication networks of heterogeneous platforms typically are sparse, i.e., far from being fully connected. If two or more routes share the same physical link, we have to decide which portion of the link bandwidth is to be assigned to each route. Once the ring and the routing have been decided, there remains to determine the best partitioning of the application data. Clearly, the quality of the final solution depends on many application and architecture parameters, and we should expect the optimization problem to be very difficult to solve.

2.2.1 Framework

Computing costs. The target computing platform is modeled as a directed graph $G = (P, E)$. Each node P_i in the graph, $1 \leq i \leq |P| = p$, models a computing resource, and is weighted by its relative cycle-time w_i: P_i requires w_i time-steps to process a unit-size task. Of course the absolute value of the time-unit is application-dependent, what matters is the relative speed of one processor versus the other.

Communication costs. Graph edges represent communication links and are labeled with available bandwidths. If there is an oriented link $e \in E$ from P_i to P_j, we let b_e denote the bandwidth of the link. It will take D_c/b_e time-units to transfer a single message of size D_c from P_i to P_j using link e.

Application parameters: computations. Let D_w be the total size of the work to be performed at each step of the algorithm. Processor P_i will accomplish a share $\alpha_i.D_w$ of this total work, where $\alpha_i \geq 0$ for $1 \leq i \leq p$ and $\sum_{i=1}^{p} \alpha_i = 1$. Note that we allow $\alpha_j = 0$ for some index j, meaning that processor P_j do not participate in the computation. Indeed, there is no reason *a priori* for all resources to be involved: the extra communications incurred by adding more processors may slow down the whole process, despite the increased cumulated speed. Also note, that we allow $\alpha_i.D_w$ to be rational: we assume the total work to be a *divisible load*.

Application parameters: communications in the ring. We arrange the participating processors along a ring (yet to be determined). After updating its data slice of size $\alpha_i.D_w$, each active processor P_i sends a message of fixed length D_c (typically some boundary data) to its successor and one to its predecessor. We assume that the successor of the last processor is the first processor, and that the predecessor of the first processor is the last processor. To illustrate the relationship between D_w and D_c, we can view the original data matrix as a large rectangle composed of D_w columns of height D_c, so that one single column is exchanged between any pair of consecutive processors in the ring. Let succ(i) and pred(i) denote the successor and the predecessor of P_i in the virtual ring. The time needed to transfer a message of size D_c from P_i to P_j is $D_c.c_{i,j}$, where the communication delay $c_{i,j}$ will be instantiated below, according to whether links are assumed to be homogeneous or heterogeneous, to be shared or dedicated.

Objective function. The total cost of a single step in the iterative algorithm is the maximum, over all participating processors, of the time spent computing and communicating:

$$T_{\text{step}} = \max_{1 \leq i \leq p} \mathbb{I}\{i\}[\alpha_i.D_w.w_i + D_c.(c_{i,\text{pred}(i)} + c_{i,\text{succ}(i)})] \tag{2.5}$$

where $\mathbb{I}\{i\}[x] = x$ if P_i is involved in the computation, and 0 otherwise. Here we use a model where each processor sequentially sends messages to its two neighbors, and we implicitly assume asynchronous receptions. There are other architectural models that could be worth investigating. In particular, we could assume that each processor is able to send both messages in parallel, which would lead to a *max* operator instead of a sum in the communication overhead of Equation 2.5.

In summary, our goal is to determine the best way to select q processors out of the p available, to assign them computational workloads, to arrange them along a ring and, if needed, to share the network bandwidths so that the total execution time per step is minimized. We denote our optimization problem as BuildRing(p, w_i, E, b_e, D_w, D_c): knowing the number of processors and their respective execution times, the existing edges and their respective bandwidths, the overall amount of work and the size of the border two successive processors are exchanging, what is the shortest achievable iteration step T_{step}? The decision problem associated to the BuildRing optimization problem is BuildRingDec(p, w_i, E, b_e, D_w, D_c, K): is there a solution to BuildRing such that T_{step} is no greater than K ?

2.2.2 Problem complexity

We now look at the complexity of our problem, with respect to the model used for representing the interconnection network. We start from a simplistic model that we gradually make more accurate and thus more complex. Along the way, we study how the complexity of our problem evolves.

Fully connected platforms with homogeneous communication links

Here, we assume that all communication links have the same characteristics and that each pair of processors is connected by a direct (physical) network link, i.e., we consider the case of fully connected platforms with homogeneous communication links. This corresponds, for example, to a cluster where processors are linked through a switch. Under these assumptions, we do not have to care about communication routing, or whether two distinct communications may use the same physical link.

Solving the optimization problem, i.e., minimizing expression (2.5), is then easy. By hypothesis, we have $c_{i,j} = c$ for all i and j, where c is a constant (it is the inverse of the bandwidth of any link). There are only two cases to consider: (i) only one processor is active; (ii) all processors are involved. Indeed, as soon as a single communication occurs, we can have several ones happening in parallel for the same

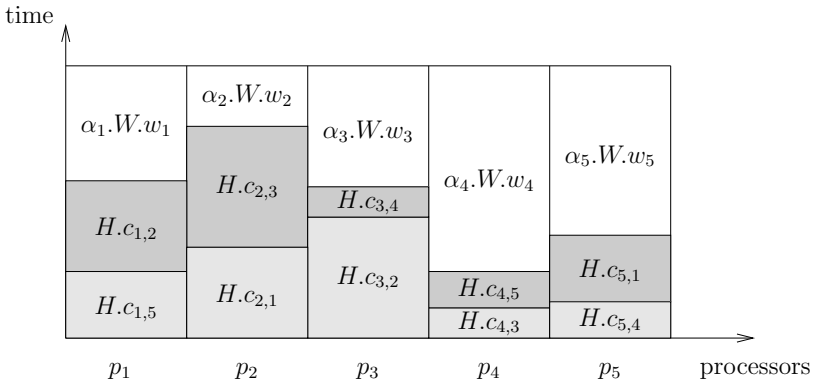


Figure 2.4: Summary of computation and communication times with $q = 5$ processors.

cost, and the best is then to divide the computing load among all resources. In the former case, the only active processor is obviously the fastest one and $T_{\text{step}} = D_w . \min_{1 \leq i \leq p} w_i$. In the latter case, the load is most balanced when the execution time is the same for all processors: otherwise, removing a small fraction of the load of the processor with largest execution time, and giving it to a processor finishing earlier, would decrease the maximum computation time. This leads to $\alpha_i.w_i = \mathsf{Constant}$ for all i, with $\sum_{i=1}^{p} \alpha_i = 1$. We derive that $T_{\text{step}} = D_w.w_{\text{cumul}} + 2D_c.c$, where $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$. We summarize these results as follows:

Proposition 1. *The optimal solution to* BUILDRING*$(p,\ w_i,\ E = \{(P_j, P_k)\}_{1 \leq j,k \leq p, j \neq k},\ b_e = \frac{1}{c},\ D_w,\ D_c)$ is*

$$T_{step} = \min \left\{ D_w . \min_{1 \leq i \leq p} w_i, D_w.w_{cumul} + 2D_c.c \right\}$$

where $w_{cumul} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$.

If the platform is given, there is an application-dependent threshold determining whether only the fastest computing resource, as opposed to all the resources, should be involved. Given D_c, the fastest processor will do all the job for small values of D_w, namely when $D_w \leq D_c . \frac{2c}{\min_{1 \leq i \leq p} w_i - w_{\text{cumul}}}$. For larger values of D_w, all processors should be involved.

Fully connected platforms with heterogeneous communication links

We are now assuming a more general model, where communication links can have different characteristics but where the interconnection network is still assumed to be complete. As the interconnection network is still complete, we assume that if a communication must occur from a processor P_i to a processor P_j, it happens along the direct physical link connecting these two processors. Therefore, as previously, we do not have to care about communication routing, or whether two distinct communications may use the same physical link.

Before formally stating the complexity of our problem on such a platform, let us consider the special case where all processors are involved in an optimal solution. All the p processors require the same amount of time to compute and communicate: otherwise, we would once again slightly decrease the computing load of the processor completing its assignment last (computations followed by communications) and assign extra work to another one. Hence (see Figure 2.4 for an illustration) we have:

$$T_{\text{step}} = \alpha_i . D_w . w_i + D_c . (c_{i,\text{pred}(i)} + c_{i,\text{succ}(i)}) \tag{2.6}$$

for all i. Since $\sum_{i=1}^{p} \alpha_i = 1$, and defining $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$ as before, we obtain:

$$\frac{T_{\text{step}}}{D_w . w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,\text{pred}(i)} + c_{i,\text{succ}(i)}}{w_i} \tag{2.7}$$

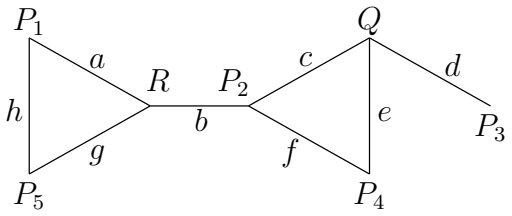


Figure 2.5: A small-size cluster.

Therefore, T_{step} will be minimal when $\sum_{i=1}^{p} \frac{c_{i,\text{pred}(i)}+c_{i,\text{succ}(i)}}{w_i}$ is minimal. This will be achieved by the ring that corresponds to the shortest hamiltonian cycle in the graph $G = (P, E)$, where each edge $e_{i,j}$ is given the weight $d_{i,j} = \frac{c_{i,j}}{w_i} + \frac{c_{j,i}}{w_j}$. Therefore, if all processors are involved, our optimization problem is equivalent to a traveling salesman problem. This hints at the NP-completeness of the problem. Formally, the following Theorem states that, as soon as the communication links become heterogeneous, our optimization problem becomes NP-complete:

Theorem 4. BUILDRINGDEC*(p, w_i, $E = \{(P_j, P_k)\}_{1\leq j,k\leq p, j\neq k}$, b_e, D_w, D_c, K) is NP-complete.*

Heterogeneous platforms

We are now considering any interconnection network. Our problem therefore becomes even harder. Formally, the decision problem associated to the BUILDRING optimization problem is once again NP-complete, and we state it for the sake of reference:

Theorem 5. BUILDRINGDEC*(p, w_i, E, b_e, D_w, D_c, K) is NP-complete.*

To really show the complexity induced by the fact that we no longer assume the interconnection network to be complete, let us consider the toy example presented on Figure 2.5. This example contains 7 processors and 8 bidirectional communication links. For the sake of simplicity, we have labeled the processors P_1 to P_5 in the order that they appear in the 5-processor ring that we construct, leaving out the other two processors Q and R. Also, links are labeled with letters from a to h instead of indices; we use b_x to denote the bandwidth of link x.

Routing. There may no longer exists a direct route between any two processors. We then assume that we can freely decide how to route messages from one processor to another. We thus have to define the communication path $\mathcal{S}_i$ ($\mathcal{S}$ stands for "successor") from P_i to $P_{\text{succ}(i)}$. Similarly, we have to define the communication path $\mathcal{P}_i$ ($\mathcal{P}$ stands for "predecessor") from P_i to $P_{\text{pred}(i)}$.

In our example, for $\mathcal{S}_1$, we arbitrarily choose to use links a and b, so that the communication path from P_1 to its successor P_2 is $\mathcal{S}_1 = \{a, b\}$. But for the path $\mathcal{P}_2$ we may use links b, g, and h, so that the communication path from P_2 to its predecessor P_1 is $\mathcal{P}_2 = \{b, g, h\}$. Here is the complete and arbitrary list of paths (note that many other choices could have been made):

- From P_1: to P_2, $\mathcal{S}_1 = \{a, b\}$ and to P_5, $\mathcal{P}_1 = \{h\}$;
- From P_2: to P_3, $\mathcal{S}_2 = \{c, d\}$ and to P_1, $\mathcal{P}_2 = \{b, g, h\}$;
- From P_3: to P_4, $\mathcal{S}_3 = \{d, e\}$ and to P_2, $\mathcal{P}_3 = \{d, e, f\}$;
- From P_4: to P_5, $\mathcal{S}_4 = \{f, b, g\}$ and to P_3, $\mathcal{P}_4 = \{e, d\}$;
- From P_5: to P_1, $\mathcal{S}_5 = \{h\}$ and to P_4, $\mathcal{P}_5 = \{g, b, f\}$.

Communication costs. It takes D_c/b_e time-units to transfer a single message of size D_c through link e. In the case of a communication using several links, the overall speed is defined by the link with the smallest available bandwidth. When there are several messages sharing the link, each of them receives a portion (to be determined) of the available bandwidth. For instance if there are two messages sharing link e, and if the first message is allocated two-thirds of the bandwidth, i.e., $2b_e/3$, then the second

message cannot use more than $b_e/3$. We assume that the portions of the bandwidth that are allocated to the messages can be freely determined by the user, the only rule is that the sum of all these portions cannot exceed the total link bandwidth. In practice, such a freedom for the routing strategy will only be available with future-generation networks like IPv6, with a suitable QoS policy framework [127]. Note however that the eXplicit Control Protocol XCP [87] does already enable to implement a bandwidth allocation strategy that complies with our hypotheses.

Let $s_{i,m}$ be the portion of the bandwidth b_{e_m} of the physical link e_m that was allocated to the path $\mathcal{S}_i$. Of course if a link e_r is not used in the path, then $s_{i,r} = 0$. Let $c_{i,\text{succ}(i)} = \frac{1}{\min_{e_m \in \mathcal{S}_i} s_{i,m}}$: then P_i requires $D_c.c_{i,\text{succ}(i)}$ time-units to send its message of size D_c to its successor $P_{\text{succ}(i)}$. Similarly, let $p_{i,m}$ be the portion of the bandwidth b_{e_m} of the physical link e_m that was allocated to the path $\mathcal{P}_i$, and $c_{i,\text{pred}(i)} = \frac{1}{\min_{e_m \in \mathcal{P}_i} p_{i,m}}$. Then P_i requires $D_c.c_{i,\text{pred}(i)}$ time-units to send its message of size D_c to its predecessor $P_{\text{pred}(i)}$.

Because of the resource constraints on the link bandwidths, the allocated bandwidth portions must satisfy the following set of inequations:

Link a: $s_{1,a} \leq b_a$	**Link b:** $s_{1,b} + s_{4,b} + p_{2,b} + p_{5,b} \leq b_b$
Link c: $s_{2,c} \leq b_c$	**Link d:** $s_{2,d} + s_{3,d} + p_{3,d} + p_{4,d} \leq b_d$
Link e: $s_{3,e} + p_{3,e} + p_{4,e} \leq b_e$	**Link f:** $s_{4,f} + p_{3,f} + p_{5,f} \leq b_f$
Link g: $s_{4,g} + p_{2,g} + p_{5,g} \leq b_g$	**Link h:** $s_{5,h} + p_{1,h} + p_{2,h} \leq b_h$

Finally, we have to write the communication costs of the communication paths. For P_1, because $\mathcal{S}_1 = \{a, b\}$, we get $c_{1,2} = \frac{1}{\min(s_{1,a}, s_{1,b})}$; and because $\mathcal{P}_1 = \{h\}$, we get $c_{1,5} = \frac{1}{p_{1,h}}$. We proceed likewise for P_2 to P_5.

Now that we have all the constraints, we can (try to) compute the α_i, $s_{i,j}$, and $p_{i,j}$ minimizing the objective function T_{step}. Equation 2.8 explicits the whole system of (in)equations which is quadratic in the unknowns α_i, $s_{i,j}$, $c_{i,j}$, and $p_{i,j}$[3]. The fact that this system is quadratic is easily explained by the fact that we are allocating bandwidths, while we are trying to optimize time, which is the inverse of bandwidths.

$$\text{Minimize} \quad \max_{1 \leq i \leq 5} \left(\alpha_i.D_w.w_i + D_c.(c_{i,i-1} + c_{i,i+1})\right) \quad \text{subject to}$$

$$\begin{cases}
\sum_{i=1}^{5} \alpha_i = 1 & & \\
s_{1,a} \leq b_a & s_{1,b} + s_{4,b} + p_{2,b} + p_{5,b} \leq b_b & s_{2,c} \leq b_c \\
s_{2,d} + s_{3,d} + p_{3,d} + p_{4,d} \leq b_d & s_{3,e} + p_{3,e} + p_{4,e} \leq b_e & s_{4,f} + p_{3,f} + p_{5,f} \leq b_f \\
s_{4,g} + p_{2,g} + p_{5,g} \leq b_g & s_{5,h} + p_{1,h} + p_{2,h} \leq b_h & \\
s_{1,a}.c_{1,2} \geq 1 & s_{1,b}.c_{1,2} \geq 1 & p_{1,h}.c_{1,5} \geq 1 \\
s_{2,c}.c_{2,3} \geq 1 & s_{2,d}.c_{2,3} \geq 1 & p_{2,b}.c_{2,1} \geq 1 \\
p_{2,g}.c_{2,1} \geq 1 & p_{2,h}.c_{2,1} \geq 1 & s_{3,d}.c_{3,4} \geq 1 \\
s_{3,e}.c_{3,4} \geq 1 & p_{3,d}.c_{3,2} \geq 1 & p_{3,e}.c_{3,2} \geq 1 \\
p_{3,f}.c_{3,2} \geq 1 & s_{4,f}.c_{4,5} \geq 1 & s_{4,b}.c_{4,5} \geq 1 \\
s_{4,g}.c_{4,5} \geq 1 & p_{4,e}.c_{4,3} \geq 1 & p_{4,d}.c_{4,3} \geq 1 \\
s_{5,h}.c_{5,1} \geq 1 & p_{5,g}.c_{5,4} \geq 1 & p_{5,b}.c_{5,4} \geq 1 \\
p_{5,f}.c_{5,4} \geq 1 & &
\end{cases} \tag{2.8}$$

Conclusion. To build up System 2.8, we have used arbitrary communication paths. There are of course many other paths to be tried. Worse, there are many other rings to be tried, made with the same processors arranged differently or using other processors. The number of processors q must be varied too. In other words, if we assume that 1) the processor selection is performed; 2) the selected processors are arranged into a ring; and that 3) the communication paths are defined, then the remaining load (and bandwidth) distribution problem is solved by a closed-form expression in the case of complete graphs and by a quadratic system in the general case.

Having a more realistic network model, first assuming that the network is no longer homogeneous, and then that it is no longer complete, each time makes our problem far more complicated. The question

[3] We did not express in Equation 2.8 the inequations stating that all the unknowns are nonnegative.

now is whether we gain anything by considering more accurate models that forbid us to derive "optimal" solutions in polynomial-time.

2.2.3 Heuristics

In this section we first present a heuristic to solve our problem on fully connected heterogeneous platforms, and then a heuristic for the general case. Then we assess the importance of using a more realistic model by comparing, on the same platforms, the solutions produced by the two heuristics.

Fully connected platforms with heterogeneous communication links

We use Equation 2.7 as a basis for a greedy algorithm which grows a solution ring iteratively. The greedy heuristic starts by selecting the best pair of processors. Then, it iteratively includes a new node in the current solution ring. Assume that we have already selected a ring of r processors. For each remaining processor P_k, we search where to insert it in the current ring: for each pair of successive processors (P_i, P_j) in the ring, we evaluate with Equation 2.7 the quality of the ring obtained by inserting P_k between P_i and P_j. We retain the processor and the pair that minimize T_{step}. Finally, among the solution which only uses the fastest processor and the rings built for each ring size in $[2, p]$, we pick the solution having the smallest T_{step}. Note that it is important to try all values of the ring size as T_{step} may not vary monotonically with the ring size.

Heterogeneous platforms

We now describe our heuristic for the general case in three steps: (i) the greedy algorithm used to construct a solution ring; (ii) the strategy used to assign bandwidth portions during the construction; and (iii) final refinements.

Ring construction. To construct the ring, we use the same greedy approach than for fully connected platforms. The difficulty here being to evaluate T_{step}. We have to resort to another heuristic to construct communicating paths and allocate bandwidth portions (explained below), in order to compute the new costs $c_{k,j}$ (path from P_k to its successor P_j), $c_{j,k}$ (the other way round), $c_{k,i}$ (path from P_k to its predecessor P_i), and $c_{k,i}$ (the other way round) which are needed to compute the new value of T_{step}. If this is carefully done, for a ring of size r, this step of the heuristic has a complexity proportional to $(p - r).r$ times the cost to compute four communicating paths. As we grow the ring until we have all p processors, the total complexity is $\sum_{r=1}^{p}(p - r)rC = O(p^3)C$, where C is the cost of computing four paths in the network.

Bandwidth allocation. Let us assume that we already have an r-processor ring, a pair (P_i, P_j) of successive processors in the ring, and a new processor P_k to be inserted between P_i and P_j. Together with the ring, we have $2r$ communicating paths, and a certain portion of the initial bandwidth has been allocated to these paths. To build the four new paths involving P_k, we reason on the graph $G = (V, E, b)$ where each edge is labeled with the remaining available bandwidth: now $b(e_m)$ is not the initial bandwidth of edge e_m, but what has been left by the $2r$ paths.

The first thing to do is to re-inject in the network the bandwidth portions used by the two communication paths between P_i and P_j (because these paths will be replaced by the new four paths). We use a simple shortest path algorithm to determine the four paths, from P_k to P_i and P_j and vice-versa. There is a subtlety here, because these four paths may share some links. The strategy that we use is the following: 1) we independently compute four paths of maximal bandwidth, using a standard shortest path algorithm [48] in G; 2) if some paths happen to share some links, we do not change the paths; instead, we use a brute force (analytical) method to compute the bandwidth portions minimizing Equation 2.7 to be allocated to each path, and we update the four path costs accordingly.

Now that we have the paths and their costs, we can compute the new value of T_{step}. (Note that from T_{step} we can derive the values of the computing workloads even if we do not need them yet.) The cost C of computing four paths in the network is $O(p + E)$.

Refinements. A concise way to describe the heuristic is the following: we greedily grow a ring by peeling off the bandwidths to insert new processors. To diminish the cost of the heuristic, we never recalculate the bandwidth portions that have been assigned to previous communicating paths. When we are done with the heuristic, we have a q-processor ring, q workloads, $2q$ communicating paths, bandwidth portions, and communication costs for these paths, and a feasible value of T_{step}.

Because the heuristic could appear over-simplistic, we have implemented two variants aimed at refining the solution. The idea for the two variants is to keep everything but the bandwidth portions and the workloads, and to recompute these each time we have inserted a new processor in the ring. In other words, once we have selected the processor and the pair minimizing the insertion cost in the current ring, we perform the insertion and we recompute all the bandwidth portions and the workloads. We keep the ring (both the processors and their ordering) and the communication paths as such. Since we know all the $2q$ paths, we can re-evaluate bandwidth portions, hence communication costs, using a global approach:

Method 1: Max-min fairness. This is the traditional bandwidth-sharing algorithm [20], which is designed to maximize the minimum bandwidth allocated to a path. Once we have computed the bandwidths portions with the algorithm, we have the communication costs, and we compute the α_i so as to equate all execution times (computations followed by communications), thereby minimizing T_{step}.

Method 2: quadratic resolution. As we mentioned earlier, once we have a ring and communicating paths, the system minimizing T_{step} is quadratic in the unknowns α_i, $s_{i,j}$, $c_{i,j}$, and $p_{i,j}$. We use the KINSOL library [129] to (approximatively) solve it.

Experimental results

The major difference between the heuristics for fully connected platforms with heterogeneous communication links and the one for the general case is that the latter takes potential link contention into account when building up the solution ring. This may look as a more clever approach. On the other hand, this may also be seen as just piling up many non guaranteed heuristics, in a desperate attempt "to do something". One may thus wonder whether, in such a case, using a more accurate network model as any practical interest. To answer this question we compare the two heuristics using the characteristics of two actual, non fully connected, platforms.

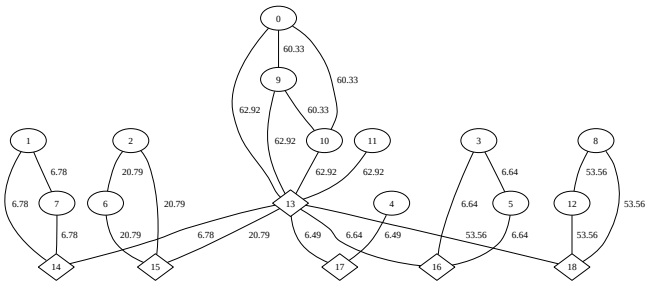


Figure 2.6: Abstraction of the Strasbourg platform.

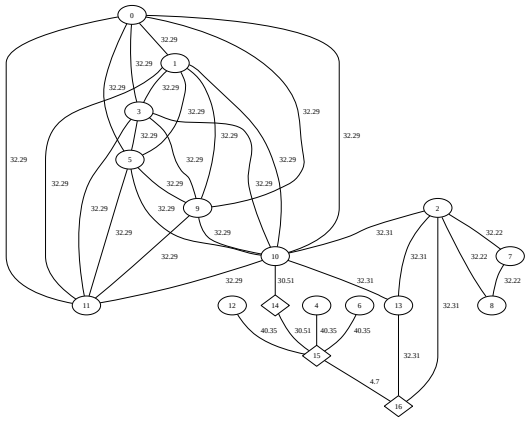


Figure 2.7: Abstraction of the Lyon platform.

Platform description. We experimented with two platforms, one heterogeneous network of workstations located in ENS Lyon and the other in the University of Strasbourg. Figure 2.6 presents an abstraction of the Strasbourg platform, which is composed of 13 computing resources (circled nodes 0 to 12) and 6 routers (diamond nodes 13 to 18). The edges are labeled with link bandwidths and the processor cycle-times are gathered in Table 2.2. Similarly, Figure 2.7 presents an abstraction of the Lyon platform, which is composed of 14 computing resources (circled nodes 0 to 13) and 3 routers (diamond

nodes 14 to 16). The edges are labeled with link bandwidths and the processor cycle-times are gathered in Table 2.3.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
0.0087	0.0072	0.0087	0.0131	0.016	0.0058	0.0087	0.0262	0.0102	0.0131

P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}
0.0072	0.0058	0.0072	0	0	0	0	0	0

Table 2.2: Processor cycle-times (in seconds per megaflop) for the Strasbourg platform.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
0.0206	0.0206	0.0206	0.0206	0.0291	0.0206	0.0087	0.0206	0.0206

P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
0.0206	0.0206	0.0206	0.0291	0.0451	0	0	0

Table 2.3: Processor cycle-times (in seconds per megaflop) for the Lyon platform.

Methodology. On our two platforms, we ran the heuristic for general platform we just described, using the max-min fairness refinement. The problem is for the heuristic designed for fully connected networks. The question is then how do we provide a (fake) completely connected network as input to this heuristic? We have to build up the communication matrix that gives the capacity of each (virtual) link between any processor pair. To construct this matrix, we simply use a shortest-paths algorithm (in terms of bandwidth), thereby simulating a complete interconnection graph. For example, with the Lyon platform, to go from processor 12 to processor 5, we use the following links: $13, 5, 4, 23$. Link 13 connects processor 12 and router 15, link 5 connects routers 15 and 14, link 4 connects router 14 and processor 10, and link 23 connects processors 10 and 5. The maximum bandwidth available is 30.51 Mb/s (these bandwidths were obtained through an *ssh* connection (constraint of encoding, etc.), which explains the relatively low value), and we store this value in the communication matrix. Clearly, the value of T_{step} achieved by the heuristic may well not be feasible as the actual network is not fully connected. Therefore, to compute the actual performance of the solution built, we keep the ring, and the communicating paths between adjacent processors in the ring, and we compute feasible bandwidth values using the max-min fairness heuristic.

Results. To compare the value of $\frac{T_{step}}{D_w}$ returned by both algorithms, we use various communication-to-computation ratios. Tables 2.5 and 2.4 show these values for each platform. The conclusions that can be drawn from these experiments is that an accurate modeling of communications has a dramatic impact on the performance of the load-balancing strategies, when communication times matter.

Note that we cannot compare our heuristics to an exhaustive search of the optimal ring. Indeed, such a research would be prohibitive as it requires not only to enumerate all processor permutations but also for each permutation to test all possible routings (as using alternate routings may decrease the amount of link sharing and thus may increase the available bandwidth).

Ratio D_c/D_w	H1: slice-ring	H2: shared-ring	Improvement
0.64	0.005825 (1)	0.005825 (1)	0 %
0.064	0.027919 (8)	0.004865 (6)	82.57%
0.0064	0.007218 (13)	0.001608 (8)	77.72%

Table 2.4: T_{step}/D_w for each heuristic on the Strasbourg platform (number between parentheses denotes the size of the corresponding ring).

Ratio D_c/D_w	H1: slice-ring	H2: shared-ring	Improvement
0.64	0.008738 (1)	0.008738 (1)	0%
0.064	0.018837 (13)	0.006639 (14)	64.75%
0.064	0.003819 (13)	0.001975 (14)	48.28%

Table 2.5: T_{step}/D_w for each heuristic on the Lyon platform (number between parentheses denotes the size of the corresponding ring).

2.2.4 Conclusion

In this section we have studied the complexity of a particular problem, the load-balancing of iterative computations, with respect to the model of the interconnection network. Unsurprisingly, the more accurate and complex the network model, the more difficult the optimization problem. Then, one may wonder whether using an accurate network model —which forbids us to derive "optimal" solutions in polynomial-time— does not make us lose more than we gain. In other words, we may wonder whether a solution, optimal for a simpler model, and roughly adapted to the general case, would not have better performance that a solution directly designed for the general case, but relying on a bunch of non-guaranteed heuristics. Using models of two actual platforms, we evaluated the heuristics we designed for two different models of interconnection networks. The comparison is meaningful as the heuristic

for the more general model is equivalent to the other heuristic when used on the simpler model. Our evaluation clearly shows that, in our context, we must use accurate platform models in order to design better heuristics and to better harness the processing power of platforms.

Chapter 3

Mapping computations and scheduling communications

We have seen in Section 2.2 what impact the communications, and their modeling, could have on the complexity of scheduling. When sheduling jobs on a platform, most of the time, one only takes into account the communications corresponding to dependences between jobs. However, the jobs may require some input data to be brought to the ressources processing them, the jobs may produce some results that should be sent back to whoever requested the computation, and the jobs themselves need to be sent to the computing ressources that are going to process them.

In this chapter we will first take into account the communications induced by the fact that the jobs and their input data must be sent to the processing nodes. In this context we will study the complexity of offline and online scheduling on star master-worker platforms (Section 3.1). As anyone would have guessed, the conclusion of this short study is that taking these communications into account can make the problem far more complicated.

The second study focuses on the general case of jobs which may share some input data (Section 3.2). After studying the different causes of the NP-completness of our problem, we aim at designing heuristics of low complexity which have performance as good as the best existing ones. In the course of this study we remark that the adaptation of simple heuristics from master-worker platforms to general platform graphs is already quite difficult. A problem with this study is that we made no assumption on the targeted applications. As a consequence, we did not have enough information to be able to design efficient sophisticated algorithms. In the next section (Section 3.3), we consider the special case of matrix multiplication on master-worker platforms when all matrices originally lay on some machine to which the result should be sent back and stored. We also suppose that the workers have limited memories. In this context we are able to derive a lower-bound on the volume of communication, and to design algorithms following the consequences of the theoretical study. While only targeting communication minimization, we were able to define an algorithm whose makespan happens to be, in practice and on average, better than those of the reference algorithms.

Several conclusions can be derived from this last study. Obviously, this is yet another example of the importance of communications in scheduling problems. Moreover, the static solution we derive is far more efficient than classical dynamic approaches. Detailed knowledge on applications often enable to inject static knowledge into algorithms, which enable to avoid classical drawbacks of pure dynamic solutions. The more precise the knowledge, the more sophisticated the algorithms, the better the performance.

3.1 Scheduling the transfer of input data on master-worker platforms

Bibliographical note: The missing details and proofs can be found in [D10].

3.1.1 Introduction

In this section, we deal with the problem of scheduling, on a heterogeneous master-worker platform, independent jobs arriving over time. We assume that this platform is operated under the *one-port* model, where the master can communicate with a single worker at any time-step. The major objective of this section is to assess the difficulty of offline and online scheduling problems under the one-port model when the communication time of the input data, and/or of the jobs, is taken into account.

Here, we only consider problems where all jobs have the same size. Otherwise, even the simple problem of scheduling with two identical workers, without paying any cost for the communications from the master, is NP-hard as it can straightforwardly be reduced to PARTITION [66]. We assume that the platform is composed of a master and p workers $P_1, P_2, \ldots, P_p$. Let c_j be the time needed by the master to send a job to P_j, and let w_j be the time needed by P_j to execute a job.

3.1.2 Offline scheduling

To be consistent with the literature [93, 26], we use the notation $\alpha \mid \beta \mid \gamma$ where:

α denotes the platform. As in the standard, we use P for platforms with identical processors, and Q for platforms with different-speed processors. We add MS to this field to indicate that we work with master-slave platforms.

β denotes the constraints. We write *online* for online problems, and r_j when there are release dates. We write $c_j = c$ for communication-homogeneous platforms, and $p_j = p$ for computation-homogeneous platforms.

γ denotes the objective. We let C_i denote the end of the execution of job i. We deal with three objective functions:

- the makespan (total execution time) $max\, C_i$;
- the maximum response time (or maximum flow) $max\, C_i - r_i$: indeed, $C_i - r_i$ is the time spent by job i in the system;
- the total completion time $\sum C_i$, which is equivalent to the sum of response times $\sum(C_i - r_i)$.

Fully homogeneous platforms

For fully homogeneous platforms, we can prove the optimality of the *Round-Robin* algorithm which processes the jobs in the order of their arrival, and which assigns them to processors in a cyclic fashion. *Round-Robin* is even optimal for the three studied objective functions: makespan, max-flow, and sum-flow. More formally:

Theorem 6. *The* Round-Robin *algorithm is optimal for the problems:*

- $P, MS \mid online, r_j, w, c \mid \max_j C_j$ *(makespan minimization);*
- $P, MS \mid online, r_j, w, c \mid \max_j (C_j - r_j)$ *(max-flow minimization);*
- $P, MS \mid online, r_j, w, c \mid \sum_j (C_j - r_j)$ *(sum-flow minimization).*

where MS indicates that the set of computational resources is organized as a master-worker platform.

We point out that the complexity of the *Round-Robin* algorithm is linear in the number of jobs and does not depend upon the platform size (contrarily to *Minimum Completion Time*).

Communication-homogeneous platforms

For communication-homogeneous platforms, we only aim at designing an optimal algorithm for the makespan minimization[1]. Intuitively, to minimize the completion date of the last job, it is necessary to allocate this job to the fastest processor (which will finish it the most rapidly). However, the other jobs should also be assigned so that this fastest processor will be available as soon as possible for the last job. We define the greedy algorithm *Scheduling Last Jobs First (SLJF)* as follows:

Initialization: Take the last job which arrives in the system and allocate it to the fastest processor (Figure 3.1(a)).

[1] Even if we stated in Section 1.3 that makespan minimization is quite meaningless for a problem with release dates! Obviously, we were not able to derive meaningful results for the two other studied objective functions.

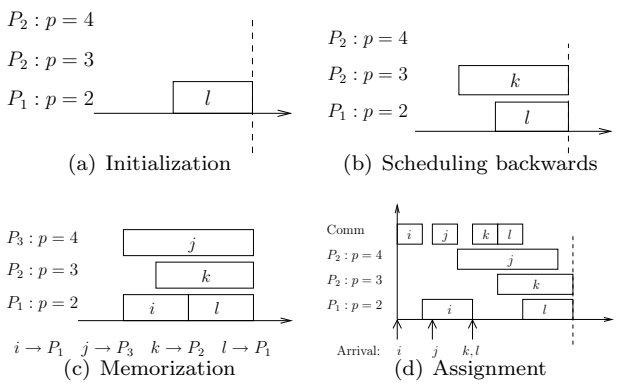


Figure 3.1: Different steps of the SLJF algorithm, with four jobs i, j, k, and l.

Scheduling backwards: Among the not-yet-allocated jobs, 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 jobs (Figure 3.1(b)).

Memorization: Once all jobs are allocated, record the assignment of the jobs to the processors (Figure 3.1(c)).

Assignment: The master sends the jobs as soon as possible according to their arrival dates, and to the processors which they have been assigned to in the previous step (Figure 3.1(d)).

Theorem 7. SLJF *is an optimal algorithm for the problem* $Q, MS \mid r_j, w_i, c \mid \max_j C_j$.

It should be stressed that, by posing $c = 0$, our approach allows to provide a new proof to a result of Barbara Simons [121].

Fully heterogeneous platforms

On fully heterogeneous platforms, the scheduling problem, $Q, MS \mid r_j, w_i, c_i \mid \max_j C_j$ is NP-hard in the strong sense. Formally, the decision problem we study is the following:

Definition 1. ***MS-hetero:*** *Given a fully heterogeneous master-worker platform, n jobs with release dates, and a deadline D, is it possible to schedule those jobs onto this platform in time D?*

The two problems are equivalent. If *MS-hetero* can be solved in polynomial time, then our problem can also be solved in polynomial time using a binary search on D on the interval $[\min_j\{\frac{n}{p} \times \max\{c_j; w_j\} + \min\{c_j; w_j\}\}, r_n + \min_j\{n \times \max\{c_j; w_j\} + \min\{c_j; w_j\}\}]$; the binary search is over the rationals, and must be stopped as soon as the gap between the upper and lower bounds becomes smaller than $\frac{1}{\text{lcm}_{1 \le i \le m}\{\beta_i, \delta_i\}}$, where $w_i = \frac{\alpha_i}{\beta_i}, \alpha_i, \beta_i \in \mathbb{N} \times \mathbb{N}^*$, $c_i = \frac{\gamma_i}{\delta_i}, \gamma_i, \delta_i \in \mathbb{N} \times \mathbb{N}^*$, α_i and β_i being prime between each other, and also γ_i and δ_i being prime between each other. Reciprocally, if we can solve our problem and find the minimum makespan D_{opt}, then for all $D \ge D_{opt}$, *MS-hetero* has a schedule, elsewhere it does not.

Theorem 8. MS-hetero *is NP-complete in the strong-sense.*

We proved this result by adapting a proof by Dutot [54].

3.1.3 Online scheduling

As *Round-Robin* is fundamentally an online algorithm, it is also optimal for the online scheduling on fully homogeneous platforms, whether the objective function is makespan, max-flow, or sum-flow minimization. As soon as either the communications or the computations are heterogeneous, there are no optimal online algorithm anymore. Table 3.1 presents the lower bounds we obtained on the competitive ratio of

any online algorithm, for the three objective functions and the three types of heterogeneous platforms. As one could have expected, we were able to derive greater lower bounds when both communications and computations are heterogeneous.

Platform type	Objective function		
	Makespan	Max-flow	Sum-flow
Fully 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.

3.1.4 Conclusion

All the problems we studied in this section were with release dates. In this framework, the problems without communications and with homogeneous communications are equivalent, from the point of view of optimality. Indeed, we can show that for any of the three studied objective functions, we can assume without loss of generality that the master is sending the jobs as soon as possible and under a *First Come, First Serve* policy. Therefore, the problem with homogeneous communications is equivalent to a problem without communications where the release date of the j-th job, r_j, is replaced by $\max_{1 \leq i \leq j} r_i + (j-i+1)c$.

For the offline version of our problem, we see that the fully heterogeneous case is strongly NP-hard for makespan minimization when we are able to design an optimal polynomial-time algorithm for the case without communication, i.e., with homogeneous communications but heterogeneous computations. For the online version, for none of the three objective functions does there exist an optimal online algorithm when communications are heterogeneous, but we have a simplistic algorithm which is optimal for the three objective functions when there are no communications and when computations are homogeneous.

This short study stresses, as one would have forecasted, that taking communications of input data into account can strongly complicate the scheduling problems.

3.2 Jobs sharing files

Bibliographical note: The missing details and proofs can be found in [D15]. Here, we will not discuss the complexity results and heuristics we derived for the master-worker version of this problem [D18].

3.2.1 Introduction

In this section we are interested in scheduling independent jobs onto collections of heterogeneous clusters. These independent jobs depend upon files (corresponding to input data, for example), and difficulty arises from the fact that some files may be shared by several jobs. Initially, the files are distributed among several server repositories. Because of the computations, some files must be replicated and sent to other servers: before a job can be executed by a server, a copy of each file that the job depends upon must be made available on that server. For each job, we have to decide which server will execute it, and to orchestrate all the file transfers, so that the total execution time is kept minimum.

An intuitive idea would be to map jobs that depend upon the same files onto the same computational resource, so as to minimize communication requirements, but such an approach may be detrimental to parallelism. On the other hand, looking for a perfect balancing of the load between the clusters may induce a large amount of communications that would slow down the overall completion of the jobs. An optimal solution would therefore certainly be a trade-off between computational load balancing and communication minimization.

This work is a follow-on of a work by Casanova, Legrand, Zagorodnov, and Berman [40, 41], work which targeted the scheduling of jobs in APST, the AppLeS Parameter Sweep Template [19]. APST is a

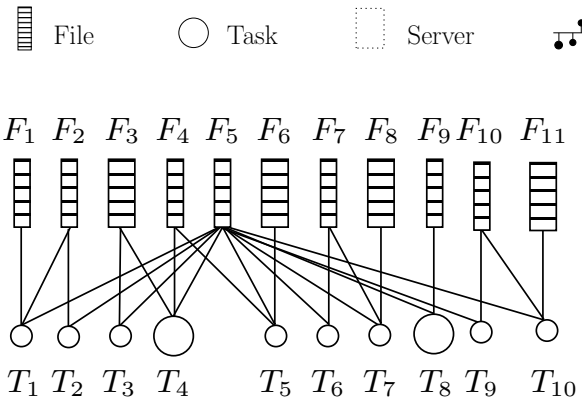


Figure 3.2: Bipartite graph gathering the relations between the files and the jobs.

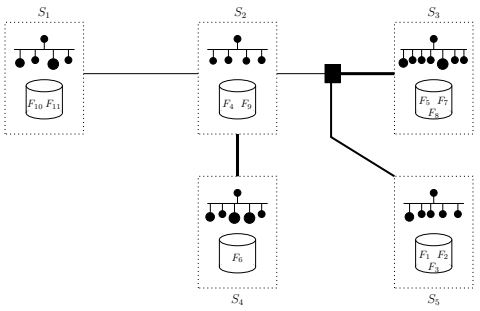


Figure 3.3: Platform graph, with the initial distribution of files to the server repositories.

grid-based environment whose aim is to facilitate the mapping of applications to heterogeneous platforms. Typically, an APST application consists of a *large* number of independent jobs with possible input data sharing. We say that the number of jobs is *large* because it is usually at least one order of magnitude larger than the number of available computing resources. MCell [126] is a typical APST application, representative of a large class of multiple Monte-Carlo simulations whose data lie in (potentially very large) files distributed across several repositories [96, 18]. See also [2] for a high-energy physics application where several jobs share data replicated in many servers.

Casanova et al. have considered three heuristics designed for completely independent jobs (no input file sharing) that were proposed in [97]. They have modified these three heuristics (originally called `min-min`, `max-min`, and `sufferage` in [97]) to adapt them to the additional constraint that input files are shared between jobs. However, all the previous references restrict to a very special case of the scheduling problem: they assume the existence of a master processor, which serves as the repository for all files. The role of the master is to distribute the files to the processors, so that they can execute the jobs. The objective for the master is to select which file to send to which worker, and in which order, so as to minimize the total execution time. Here, we deal with the most general instance of the scheduling problem: we assume a fully decentralized system, where several servers, with different computing capabilities, are linked through an interconnection network. To each server is associated a (local) data repository. Initially, the files are stored in one or several of these repositories (some files may be replicated). After having decided that server S_i will execute job J_j, the input files for J_j that are not already available in the local repository of S_i will be sent through the network. Several file transfers may occur in parallel along disjoint routes.

The rest of this study is organized as follows. The next section (Section 3.2.2) is devoted to the precise and formal specification of our scheduling problem, which we denote as JSFDR (*Jobs Sharing Files from Distributed Repositories*). Next, in Section 3.2.3, we give an overview of some complexity results on our problem. After assessing its theoretical difficulty, we move to the design of polynomial time heuristics to solve it. In Section 3.2.4, we show how one can extend the `min-min` heuristic to our decentralized framework. This extension turns out to be surprisingly difficult, and we detail both the problems encountered, and the solution provided. As pointed out, the number of jobs to schedule is expected to be very large, and special attention should be devoted to keeping the cost of the scheduling heuristics reasonably low. Therefore, in Section 3.2.5 we deal with the design of low-cost polynomial-time heuristics retaining the good performances of the `min-min` variants. We report some simulation data in Section 3.2.6. Finally, we close with some concluding remarks in Section 3.2.7.

3.2.2 Framework

Here we formally state the optimization problem to be solved.

Application model. The problem is to schedule a set of n independent jobs $\mathcal{T} = \{J_1, J_2, \ldots, J_n\}$ with job J_j having size p_j. There are no dependence constraints between the jobs, so they can be viewed as

independent (a job never takes as input the result of the computation of another job).

The execution of each job depends upon one or several files, and a given file may be shared by several jobs. Altogether, there are f files in the set $\mathcal{F} = \{F_1, F_2, \ldots, F_f\}$. The size of file F_i is f_i, $1 \leq i \leq f$. We use a bipartite application graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to represent the relations between files and jobs. The set of nodes in the graph $\mathcal{G}$ is $\mathcal{V} = \mathcal{F} \cup \mathcal{T}$, and there is an edge $e_{i,j} : F_i \rightarrow J_j$ in $\mathcal{E}$ if and only if job J_j depends on file F_i. Intuitively, the files F_i with $e_{i,j} \in \mathcal{E}$ contain data needed as input for the execution of job J_j. The processor that will have to execute job J_j will need to receive all the files F_i with $e_{i,j} \in \mathcal{E}$ before it can start the execution of J_j. See Figure 3.2 for a small example, with $f = 11$ files and $n = 10$ jobs. For instance, job J_1 depends upon files F_1, F_2, and F_5, and file F_5 is an input to all jobs J_1 to J_{10}. In the figure, we draw jobs and files of different sizes to symbolically represent their heterogeneity. To summarize, the bipartite *application graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node in $V = \mathcal{F} \cup \mathcal{T}$ is weighted by f_i or p_j, and where edges in $\mathcal{E}$ represent the relations between the files and the jobs, models all the information on the application.

Platform model. The jobs are scheduled and executed on an heterogeneous platform composed of a set of *servers*, which are linked through a platform graph $\mathcal{P} = (\mathcal{S} \cup \mathcal{R}, \mathcal{L})$ made of communications links and of routers. Each node in $\mathcal{S} = \{S_1, \ldots, S_s, \}$ is a server, each node in $\mathcal{R} = \{R_1, \ldots, R_r\}$ is a router, and each link $l = (u, v)$ represents a communication link from $u \in \mathcal{S} \cup \mathcal{R}$ to $v \in \mathcal{S} \cup \mathcal{R}$, i.e., a communication link can connect two servers, two routers, or a router and a server. We assume that the graph $\mathcal{P}$ is connected, therefore there is a path between any server pair. By default, we assume that all links are bidirectional, hence $\mathcal{P}$ is undirected, but we could easily deal with oriented links.

Each server $S_i = (D_i, C_i)$ is composed of a local data repository D_i, associated to a local computational cluster C_i. The files needed by the computations (the jobs) are stored in the data repositories. We assume that a file may be duplicated, and thus simultaneously stored on several data repositories. We make no restriction on the possibility of duplicating the files, which means that each repository is large enough to hold a copy of all the files. See Figure 3.3 for an example of platform.

A cluster can process a job only if the corresponding data repository contains all the files that the job depends upon. With the previous notations: for cluster C_i to be able to process job J_j, repository D_i must hold all files F_k with $e_{k,j} \in \mathcal{E}$. Therefore, before C_i can start the execution of J_j, the server S_i must have received from the other server data repositories all the files that J_j depends upon, and which were not already stored in D_i.

Communication model. For communications, we use the one-port model: at any given time-step, there are at most two communications involving a given server, one sent and the other received. Furthermore, we make the assumption that the routing inside the platform graph is fixed: files sent from one server to another will always follow the same path.

As for the cost of communications, consider first the case of two *adjacent* servers, S_i and S_j, in the platform graph. We say two servers are adjacent if the communication path connecting S_i and S_j does not pass through intermediate servers (but it can use routers). For example, in Figure 3.3, servers S_3 and S_2 are adjacent but S_3 and S_1 are not, as a communication between S_3 and S_1 must go through server S_2. Suppose that server S_i sends the file F_j (stored in its repository D_i) to an adjacent server S_k, using a routing path composed of the communication links $l_1, l_2, ..., l_{x-1}, l_x$. Let b_y denote the bandwidth of link l_y. Then the bandwidth available for the communication path is the minimum of the bandwidths of the different links, so that $\frac{f_j}{\min_{1 \leq y \leq x} b_y}$ time-units are required to send the file. Next, for communications involving intermediate servers, we use a store-and-forward model between servers: we route the file from one server to an adjacent one, leaving a copy of the file in the data repository of each intermediate server. The communication cost is the sum of the costs of the adjacent communications. Leaving copies of transferred files on intermediate servers multiplies the number of potential sources for each file and is likely to accelerate the processing of the next jobs, hence the store-and-forward model seems quite well-suited to our problem. The communications through intermediate servers, along with the store-and-forward model for these communications, allow us to model systems where, for security or privacy reasons, servers must only use pre-defined communication channels. The fact that this store-and-forward mechanism is used or not depends on the platform characteristics. For example, if all communications are between adjacent servers, the store-and-forward mechanism will not be used.

Finally, we suppose that when the necessary files are on a server data repository, they are available

for free on its cluster. In other words, we assume no communication time between a cluster and its associated data repository: the cost of intra-cluster messages is expected to be an order of magnitude lower than that of inter-cluster ones. We also assume that the only communication costs are due to the communication of files. Indeed, we consider no migration cost for assigning a job to a cluster. In another model, one could imagine that the code of a job originally lies on a data repository and that it should also be sent to the data repository linked to the cluster the job is assigned to. This model can easily be embedded in ours as one only need to add to our bipartite graph of relations between files and jobs some "virtual files" representing job codes.

Computation model. As for computation costs, each cluster C_i is composed of heterogeneous processors. More precisely, C_i gathers m_i processors $C_{i,k}$, $1 \leq k \leq m_i$. The speed of processor $C_{i,k}$ is $s_{i,k}$, meaning that $p_j/s_{i,k}$ time-units are needed to execute job J_j on $C_{i,k}$. A coarser approach is to view cluster C_i as a single computational resource of cumulative speed $\sum_{k=1}^{m_i} s_{i,k}$. We easily model the situation where a given server is composed of a single data repository but has no computational capability: we simply create a (fake) cluster of null speed.

Objective function. The objective is to minimize the total execution time (or makespan). The execution is terminated when the last job has been completed. The schedule must decide which jobs will be executed by each processor of each cluster, and when. It must also decide the ordering in which the necessary files are sent from server data repositories to server data repositories. We stress three important points:

- Some files may well be sent several times, so that several clusters can independently process jobs that depend upon these files.
- A file sent to some data repository remains available on it for the rest of the schedule; so, if two jobs depending on the same file are scheduled on the same cluster, the file must only be sent once.
- Initially, a file is available on one or several well-identified servers. But when routing a file from one of these servers to another non adjacent one, a copy is left on each intermediate server data repository. Hence all the intermediate servers (if any), in addition to the server which is the final destination of the file in the communication, become potential sources for the file.

3.2.3 Complexity

Most scheduling problems are known to be difficult and the JSFDR optimization problem is no exception. What is interesting are the reasons why this problem is NP-complete.

Heterogeneity may be a cause of NP-completeness. Here, heterogeneity may come from several sources: files or jobs can have different weights, while clusters or links can have different speeds.

1. The decision problem associated to the instance with no files and two single-processor clusters of equal speed already is NP-complete: in that case, JSFDR reduces to the scheduling of independent jobs on a two-processor machine, which itself reduces to the Partition problem [66] as the jobs have different weights. Hence, the heterogeneity of jobs causes the problem to be NP-complete.
2. Mapping equal-size files and equal-size jobs on a single server platform with two heterogeneous processors and two links of different bandwidths is NP-hard too [D18, B7, C13]. Hence, the heterogeneity of processors and communication links causes the problem to be NP-complete.

In fact, just to decide where to move the files so as to execute the jobs is already a difficult combinatorial problem, even in the un-weighted version where all files have same size and all communication links have same bandwidth. We even assume that all jobs have zero weight, or equivalently that all clusters have infinite speed. In this simple case, our problem becomes a mapping problem: we have to map the jobs to the clusters and, for each job, we have to gather its required set of files (that are its inputs) on the repository of the server it is mapped to, with the objective to minimize the number of communications steps (all communications have unit-time, but independent communications, i.e., involving distinct senders and distinct receivers, may well take place in parallel). Formally, we state the decision problem associated to this very particular instance of JSFDR as follows:

Definition 2 (JSFDR-Move-Dec($\mathcal{G}, \mathcal{P}$,K))**.** *Given a bipartite application graph $\mathcal{G} = (\mathcal{F} \cup \mathcal{T}, \mathcal{E})$, a platform graph $\mathcal{P} = (\mathcal{S}, \mathcal{L})$, assuming:*

- *uniform file sizes ($f_i = 1$),*
- *homogeneous interconnection network ($b_i = 1$),*
- *zero processing time ($p_i = 0$ or $s_j = +\infty$),*

and given a time bound K, is it possible to schedule all jobs within K time-steps?

Theorem 9. JSFDR-Move-Dec*($\mathcal{G}, \mathcal{P}$,K) is NP-complete.*

We stress that this last NP-completeness result holds for an un-weighted version of the original problem, which proves that the difficulty already lies in the combinatorics of allocating and duplicating files, in the absence of any heterogeneity neither in the application (files and jobs) nor in the platform (processors and links). The general version of the *JSFDR* problem involves weights in jobs, in files, and in the platform graph. We believe that approximation algorithms are not likely to be feasible (just as it is the case for the graph partitioning problem). Therefore, in the next sections, we design polynomial heuristics to solve the JSFDR problem, and we assess their performance through extensive simulations.

3.2.4 Adapting the `min-min` scheme

Considering the work of Casanova et al. [40, 41] for master-worker systems with a single server, we start by adapting the `min-min` scheme. In the next section we will introduce several new heuristics, whose main characteristic is a lower computational complexity than that of the `min-min` scheme.

Principle of the `min-min` scheme. The principle of the `min-min` scheme is quite simple:

- While there remain jobs to be scheduled do
  1. for each job J_k that remains to be scheduled, and each processor $C_{i,j}$ in the system, evaluate the Minimum Completion Time (MCT) of J_k if mapped on $C_{i,j}$;
  2. pick a couple $(C_{i,j}, J_k)$ with minimum MCT, and schedule job J_k on processor $C_{i,j}$.

The difficulty with this heuristic is to evaluate the Minimum Completion Time (MCT). When trying to schedule a job on a given processor, one has to take into account which required files already reside in the corresponding repository, and which should be routed through the network of servers. If a file that must be routed has already been replicated on several servers, we must decide which server will be the source of its transfer. Once the source of each transfer is decided, it is straightforward to determine which communications should take place. However, scheduling these communications is an NP-complete problem in the general case, as we will recall.

Once we have decided how to schedule the communications, there only remains to evaluate the computation time. To evaluate the MCT of a job on a server, we should schedule the jobs on its cluster *processors*. However, we heuristically decide to view a whole cluster as a single processor whose processing power is the sum of the processing powers of the cluster processors (the coarse-grain approach alluded to in Section 3.2.2). The start date for the execution of the new job is set as the latest date between (1) the arrival of the last of the necessary files, and (2) the end of the (coarse evaluation of the) computation of the jobs assigned so far to the cluster. With this simplified coarse-grain view for computations, the only problem we are left with is the scheduling of the communications.

Scheduling the communications. As the jobs are allocated on processors one at a time, we must schedule a set of communications all having the same destination (namely the server that will execute the job). In the 1-port model, if the routing in the platform graph is not fixed, scheduling a set of communications is already NP-hard [D15] in our context. As the routing is usually decided by table look-up, one can assume the routing to be fixed. But when trying to schedule the communications required for a task T_k, one must take into account that the communication links are already used at certain time slots due to previously scheduled communications. Even under a fixed routing, this also leads to an NP-complete problem [D15]. Therefore, we (heuristically) decide to use simple greedy algorithms

to schedule the communications required to send the necessary files to the server which we want to assign the new job to. The first algorithm uses an *insertion scheduling* scheme, and the second one always schedules a new communication *after* any communication already scheduled on the same links.

In the following, we call:

- *transfer*: the communication, through the network of servers, of a file from the source server (which stores the file in its repository) to the destination server (where we attempt to schedule the new job);
- *adjacent communication*: the communication between two adjacent intermediate servers; therefore, a transfer is potentially made up of several adjacent communications.

A first idea is to memorize for each link the date and length of all communications already scheduled, and to use an *insertion scheduling* scheme to schedule any new communication in the first possible time slot (first fit approach). This scheme should be rather efficient but may be very expensive. A second idea is to schedule new communications as soon as possible but always *after* communications using the same links. In both cases, if a file is available on several servers, we heuristically decide to bring it from the closest server, i.e., from the server from which the file communication time is minimal (considering an empty platform). Also, in both variants the adjacent communications are scheduled in decreasing order of the distance of their destination to the final destination D of all transfers. In the insertion scheduling variant, the adjacent communications are scheduled in decreasing order of their duration and scheduled in the first possible time slot. In the simpler variant, the adjacent communications are scheduled in increasing order of their availability dates, and a new adjacent communication from R to S is always scheduled after the last communications involving R as sender and/or S as receiver.

Of course, these algorithms do not always compute the optimal solution (cf. [D15] for an example).

With *insertion scheduling*, the complexity of the communication scheduling is $O(\Delta T m \Delta \mathcal{P})$ where we denote by ΔT the maximum number of files that a job depends upon, and by $\Delta \mathcal{P}$ the diameter of the platform graph.

Without insertion scheduling, the complexity of the communication scheduling is $O(\Delta \mathcal{P} \Delta T \log \Delta T)$ In the rest of this section, we denote by O_c the complexity of communication scheduling with our heuristics. Then, O_c will be equal to either of the two above formulas, depending on the version of the algorithm chosen.

Complexity. The complexity of the `min-min` heuristic is then $O(sn(nO_c + s|\mathcal{E}|) + n \max_{1 \leq i \leq s} m_i)$. The last term in this expression comes from scheduling the jobs on the clusters. Indeed, once the communications are scheduled, we have for all jobs the availability dates of the files they depend upon. Then on each cluster we greedily schedule the jobs on the processors. Among the jobs of lowest availability date, we schedule the largest one on the processor on which it will have the minimum completion time.

The `sufferage` variant of the `min-min` scheme. This variant of `min-min` sometimes delivers better schedules, but its complexity is slightly greater [40, B7, C13]. Instead of choosing a couple (S_j, J_i) with minimum MCT, the chosen job J_i is the one which will be the most penalized if not mapped on its most favorable processor, but on its second most favorable, i.e., the job with the greater difference between its two minimum MCTs. This job is then mapped on a server S_j which minimizes its MCT.

3.2.5 Heuristics of lower complexity

As appealing as the `min-min` scheme could be because of the quality of the scheduling that it produces [40, D18], its computational cost is huge and may forbid its use. Therefore, we aim at designing heuristics which are an order of magnitude faster, while trying to preserve the quality of the scheduling produced. The `min-min` scheme is especially expensive as, each time it attempts to schedule a new job, it considers for each cluster all the remaining jobs and compute their MCTs. On the opposite, we worked on solutions where we only consider a single job candidate per cluster. This leads to the following scheme:

- While there remain jobs to be scheduled do
  1. for each cluster C_i pick the "best" candidate job J_k that remains to be scheduled;
  2. pick the "best" couple (C_i, J_k) and schedule J_k on C_i.

The whole heuristic then relies on the definition of the "best" candidate. For that, we design a *cost* function used to estimate the minimum completion time of a job on a given server. We have designed two types of heuristics: static ones where *costs* are estimated once and for all; and dynamic ones where *costs* are reevaluated as mapping and scheduling decisions are taken.

Static heuristics

In our static heuristics, for each cluster we first build a list of all jobs sorted by increasing *cost* function. Then, each time we schedule a new job: 1) we define as local candidate for cluster C_i the job which has lowest *cost* on C_i and has not already been scheduled; 2) among all the local candidates we take the one of lowest *cost* and we assign it on the corresponding cluster; 3) we schedule the necessary communications and the computation as we did for the `min-min` scheme.

Cost function. We define the *cost* of a job J_i on a server S_j as an evaluation of the completion time of J_i on S_j. Following what has been previously done for the `min-min` heuristic, the completion time could be defined as the sum of the time needed to send the required files to S_j, plus the time needed by the cluster C_j to process the job, when the cluster is seen as a single computational resource, and when no other communications or computations take place on the platform. Actually, we speed-up the computation of *costs* by approximating the communication time either by the sum of the transfer times (pessimistic over-approximation by sequentializing all transfers) or by the "max" of these transfer times (optimistic under-approximation by considering that all transfers take place in parallel).

In the following we denote by `static` the static heuristic presented above. Its complexity is an order of magnitude less than the complexity of the `min-min` scheme as we no longer have a n^2 term: $O\left(s^2\Delta\mathcal{P} + s|\mathcal{E}|nO_c + n(\log n + p)\right)$.

Variant `readiness`: highest priority to ready jobs. In the basic version of the heuristic, jobs are selected by following strictly the increasing order on the job *costs*. In the `readiness` variant, each time we try to schedule a new job, we consider the next job in this order, *unless* there is a job J_i which is *ready* for a server S_j, in which case we immediately schedule J_i on S_j. A job is called *ready* for a server, if the server repository holds all the files that the job depends upon. So, a ready job can be scheduled at no communication cost. This `readiness` variant has been proposed in [111] under the name *JobDataPresent*. Maintaining lists of ready jobs costs $O(s|\mathcal{E}|)$.

Variant `mct`: postponing the mapping of jobs to servers. In the `static` heuristic, a job is mapped on a server on which its *cost* is minimal. Following ideas in [D18], we only use the *costs* to determine in which order the jobs are considered. Thus, we sort, for each server, the jobs by increasing *cost*. The jobs are still scheduled one-at-a-time. When we want to schedule a new job, on each server S_i we evaluate the completion time of the job of lowest *cost* which has not yet been scheduled. The completion time evaluation is identical to the equivalent evaluation performed by the `min-min` heuristic. Then we pick the pair job/server with the minimum completion time. We choose to include by default the `readiness` policy in this variant. This way, we obtain our `static+mct` heuristic. The overall complexity of `static+mct` is: $O\left(s^2\Delta\mathcal{P} + sn\log n + s^2|\mathcal{E}| + nsO_c + n(n\log n + p)\right)$.

The expected impact of the `mct` refinement is a better load-balancing of jobs onto servers, which is especially important in situations where the basic version might have mapped all jobs on the same server. In some peculiar situations (e.g., low cost communications and one server much faster than the others), the optimal solution may well be to execute all jobs on the same server. However, in most practical situations, the execution of jobs will be distributed among several servers, and the `mct` variant accounts for balancing the server loads as new mapping decisions are being taken.

Dynamic heuristics

In our static heuristics, we first define the order in which the jobs are considered, and all the other scheduling decisions (communication definition and scheduling) are implied by this original order. However, this order is based on a *cost* which is only truly relevant when there is a single job in the system. Indeed, the *cost* formula does not take into account the fact that several jobs may need the same file. Thus it misses the possibility that new sources may have been created, as the execution proceeds, for

these files. To remedy this flaw, we introduce a more dynamic scheme, while trying to conserve a low complexity.

We want a dynamic *cost* function which returns us the same estimation for the completion time of a job T on a server S as the original *cost* function would return if it was taking into account the knowledge, at the time of its invocation, of:

1. which files T depends upon have been duplicated and where;
2. the amount of jobs previously mapped on S.

The first type of knowledge enables to choose transfer sources which are closer to the server S, thereby decreasing the needed communication time. The problem is to implement such a function for the lowest possible complexity. We want to precompute this dynamic *cost* as much as possible, and not just call the original *cost* function each time we need to know the *costs*. Therefore, each time a new communication is decided involving a file F, we recompute the *closest* source of F for each server S_i. Then each time the *closest* source for a file F_k and a server S_j changes, we update the *cost* of J_i on S_j if and only if job J_i depends upon file F_k, i.e., $e_{k,i} \in \mathcal{E}$. Any of these updates is done in constant time if we over-approximate the communication time (sum of the transfer times from which we subtract the decrease of the transfer time), and costs $O(\Delta T)$ if we under-approximate it (maximum over all the transfer times). The overall complexity of maintaining our dynamic *costs* is then $O(s^2(\Delta\mathcal{P} + \log s) + s^2|\mathcal{E}|u)$ where $u = 1$ with over-approximation and $u = \Delta T$ otherwise.

Once we have defined the dynamic *cost*, we have to decide how to use it to select the new job(s) to be scheduled. We have the choice either to pick a single new job at a time or to pick a set of k jobs. The former scheme is in spirit closer to the `min-min` heuristic. The latter scheme may be less expensive. In both versions, we once again target a low complexity.

In heuristic `dynamic1` we maintain on each server a *heap* of the job *costs*. Then, on each server, the selection of the job of lowest *cost* is done in constant time. However, each time we update a *cost*, the removal of an element from the heap and the addition of a new one cost $O(\log n)$. The overall complexity of this heuristic is: $O\left(s(n \log n) + nO_c + s^2(\Delta\mathcal{P} + \log s + |\mathcal{E}|(u + \log n)) + np\right)$.

In heuristic `dynamic2`, to further decrease the complexity of the selection of the "best" job candidate, we select, on each server, k jobs of lowest *costs*, instead of only 1. Such a selection can be realized in linear time in the worst case [28, 48]. The selection process will then be quicker at the expense of its quality. As a job may appear in the "least expensive" set of different servers, we sort the ks couples (job, server) that we obtain according to their *costs*, and we pick the k distinct jobs of lowest *costs*. The overall complexity of this heuristic is: $O\left(\frac{n}{k}s(n + k\log(ks)) + nO_c + s^2(\Delta\mathcal{P} + \log s + |\mathcal{E}|u) + n(\log n + p)\right)$.

3.2.6 Simulation results

In order to compare our heuristics, we have simulated their execution on randomly built platforms and graphs. We have conducted a large number of simulations, which we summarize in this section.

Simulated platforms. The *platform graphs* are composed of 7 servers (see below for larger platforms). A graph is either a clique, a random tree, or a ring. For the clique platforms, the fixed routing either follows a shortest communication path rule (the platform is then denoted *clique-time*) or always use the direct single-link route connecting two servers (denoted *clique-distance*). The latter corresponds to servers only connected through WAN links in the vision of [37, Fig. 2].

We have recorded the computational power of different computers used in our laboratories (Lyon and Strasbourg). From this set of values, we randomly pick values whose difference with the mean value was less than the standard deviation. This way we define realistic and heterogeneous *clusters* randomly containing 8, 16, or 32 processors. The bandwidths of the *communication links* are randomly chosen along the same principles as the processors speeds, using values measured in our laboratories.

The absolute values of the communication link bandwidths or of the processors speeds have no meaning (in real life they must be pondered by application characteristics). We are only interested by the relative values of the processors' speeds, and of the communication links' bandwidths. Therefore, we normalize processor and communication characteristics. Also, we arbitrarily impose the *communication-to-computation cost ratio*, so as to model three main types of problems: computation intensive (ratio=0.1), communication intensive (ratio=10), and intermediate (ratio=1).

Application graphs. We ran the heuristics on four types of application graphs: 1) Fork: each graph contains 70 fork graphs, where each fork graph is made up of 21 or 22 jobs depending on a single and same file; 2) Two-one: each job depends on exactly two files: one file which is shared with some other jobs, and one un-shared file; 3) Partitioned: the graph is divided into 20 chunks of 75 jobs, and in each chunk each job randomly depends on 1 up to 10 files; the whole graph contains at least 20 different connected components; 4) Random: each job randomly depends on 1 up to 50 files. In each case, the size of the files and jobs are randomly and uniformly taken between 0.5 and 5.

Our objective is to use graphs *representative* of a large application class. The fork graphs represent embarrassingly parallel applications. The two-one graphs come from the original papers by Casanova et al. [40, 41]. The partitioned graphs deal with applications encompassing some regularity. The random graphs are for totally irregular applications. Each graph contains 1,500 jobs and 1,750 files, except for the fork graphs which also contain 1,500 jobs but only 70 files. In order to avoid any interference between the graph characteristics and the communication-to-computation cost ratio, we normalize the sets of jobs and files so that the sum of the file sizes equals the sum of the job sizes times the communication-to-computation cost ratio. The initial distribution of files to server is built randomly.

Results

Small platforms. Table 3.2 summarizes all the experiments (more detailed results can be found in [D15]). In this table, we report the performance of the heuristics, together with their cost (i.e., their CPU time). This is a summary of 48,000 random tests (1,000 tests over all four application graph types, four platform graph types, and three communication-to-computation cost ratios). Each test involves 44 heuristics: all the heuristics previously described with their variants plus the naive `randomjob` heuristic, which randomly picks the local candidate (the same for all clusters) but uses the same optimizations and scheduling schemes than the other heuristics. In other words, only the choice of the next job to be executed is random. In contrast, the choice of the server where to execute it (which is crucial) is determined by the previous mapping policies. This heuristic is combined with the variant `mct`. Each heuristic was tested with insertion scheduling of the communications (`insert`) or without.

For each of the 48,000 tests, we compute the ratio of the performance of all heuristics over the best heuristic for the test, which gives us a *relative performance*. The best heuristic differs from test to test, which explains why no heuristic in Table 3.2 can achieve an average relative performance exactly equal to 1. In other words, the best heuristic is not always the best of each test, but it is closest to the best of each test on average. The optimal relative performance of 1 would be achieved by picking, for any of the 48,000 tests, the best heuristic for this particular case. (For each test, the relative cost is computed along the same guidelines, using the fastest heuristic.)

The `min-min` variants, `min-min` and `sufferage`, achieve, on average, similar performance. Their variants where the communications are scheduled using an insertion scheduling scheme are the best heuristics, but only by a slight margin.

The basic versions of our heuristics are far quicker than the `min-min` versions but at the cost of a great loss in the quality of the schedules produced (at least two times worse). As we had already observed in [D18], the `readiness` variant has a significant impact on performance. However, as it does not include a load-balancing scheme, with the `readiness` policy most jobs may end-up mapped on a single cluster, inducing a large imbalance (this is best exemplified by `static+readiness` having worse performance than `static` on *fork* graphs). The `dynamic` heuristics contain both the `readiness` policy and a load-balancing rule. Therefore, it may be surprising that the `static` variants are all better than their `dynamic` counterparts. This is explained by the fact that the `dynamic` heuristics do not take into account link contentions. Then, in order to balance the load these heuristics can map jobs on servers which are connected to other servers by congested communication links. In practice, the jobs will not be executed simultaneously and thus the load will not be balanced. This suggests to use the `readiness` variant in conjunction with the `mct` variant which naturally balances the load. The `mct` variant greatly improves the quality of our heuristics while their costs remain very low.

As foreseen, the larger the packet size in `dynamic2`, the lower the cost and the worse the performance, and `dynamic2` always delivers worse performance than `dynamic1`. Also, the basic `randomjob` is, not surprisingly, the worst heuristic with a relative performance of 130. The pessimistic evaluation of the communication costs almost always delivers better performance than the optimistic one (`max` variant) as most configurations are congestion-proned.

Our heuristics have the same behavior whatever the communication to computation ratio. Furthermore, the `mct` variants of `dynamic1`, `randomjob` and `static+readiness` have almost the same relative performance whatever this ratio is, which shows their robustness.

We also ran the heuristics with the insertion scheduling heuristic for communication scheduling (rather than with the greedy scheduling as previously). As predicted, the quality of results significantly increased. The overhead is prohibitive for the `min-min` variants. Surprisingly, this overhead is reasonable for our heuristics. For example, the version `dynamic1+mct+insert` produces schedules which are 3% *better* than those of the original `min-min` and it produces them 288 times more quickly! The `randomjob+mct+insert` heuristics performs even better: it achieves similar performance even quicker.

Study on larger platforms. We compared our best heuristics on larger platforms to assess the impact of the platform sizes and to see how our results scale (see Table 3.3). We studied two types of platforms: 1) 50 server platform graphs with 25,000 jobs and 32,000 files (1,250 files for *forks*); 2) 100 server platform graphs with 50,000 jobs and 62,500 files (2,500 files for *forks*). The two sets of platforms have the same average load per server, which is 2.5 times larger than on our original 7 server platforms. These simulation configurations are too large for the `min-min` heuristics. The 50 server configuration is the largest we can run the `insert` variant on, knowing our simulation testbed.

On these larger configurations, we mainly notice that: 1) The `mct` variant (with `readiness`) is once again essential; 2) The `mct` variants of `dynamic` heuristics have the forecasted relative behavior: `dynamic1` is slower but better than `dynamic2+010` which is slower but better than `dynamic2+100`; 3) `randomjob+mct+insert` is significantly worse than `dynamic1+mct+insert` when it was helped by the small size of the 7 server configurations (most significantly on the two-one application graphs); `dynamic1+mct` is significantly better than `randomjob+mct`, except for fork graphs where they have similar results.

Summary. We have reported in Table 3.3 the heuristics that give the best results given some time limit. We can see that, except `min-min` and `sufferage` that rapidly become too expensive, a good choice is to use `dynamic1+mct` (or even `randomjob+mct`). If we can afford it, scheduling communications with an insertion scheduling scheme greatly improves the quality of the produced schedules.

In our simulation framework, on 7 server platforms, `min-min` takes on average 38 seconds to build a schedule, and `dynamic1+mct` only 0.1 second. On 50 server platforms, `dynamic1+mct` builds a schedule on average in 511 seconds, when it would cost `min-min` more than 48 hours (for 25,000 jobs).

3.2.7 Conclusion

Here, we have dealt with the problem of scheduling a large collection of independent jobs, that may share input files, onto collections of distributed servers. We have seen how one can extend the well-known `min-min` heuristic to the new framework; this turned out to be more difficult than expected because of the complexity of communication scheduling problems. We have succeeded in designing a collection of new heuristics which have reasonably good performance but whose computational costs are orders of magnitude lower than `min-min`. The best heuristics were obtained by combining the `readiness`, `mct`, and `insert` variants. Specifically, on small platforms, the heuristic `randomjob+mct+insert` produces schedules whose makespan is only 4% longer to those produced by the best variant of `min-min` (namely `sufferage+insert`), and produces them 371 times faster than the fastest variant of `min-min` (the basic `min-min`). On larger platforms, the running time of the `min-min` heuristics and of the `insert` variants become prohibitive. Then, `dynamic1+mct` becomes the heuristic of choice.

The fundamental conclusion of this study is that what really matters is not the order in which jobs are considered and mapped, but that jobs which do not induce communications should be considered first (`readiness` variant) and that among the candidate jobs, if any, the one chosen should be the one finishing first (`mct` variant). Hence the very good performance of the heuristic `randomjob+mct`. Although the heuristics presented here where designed in an offline framework, the `dynamic` and `randomjob` heuristics could be used as such in an online setting, but should be refined to prevent starvation (e.g., by cleverly decreasing a job cost as its waiting time increases).

All the heuristics discussed here or in the original work of Casanova, Legrand, Zagorodnov, and Berman [40, 41], are *local* heuristics, i.e., they take mapping and scheduling decisions only using a local view of the problem and not a global one. It would a priori be better to use a global approach.

Table 3.2: Relative performance and cost of the heuristics: basic version and `mct` variants, with or without communication scheduling with insertion scheduling (`insert`). Averages on all 48,000 configurations; standard deviations are in parentheses.

Heuristic	Basic version		`mct` variant		`insert` variant		`mct+insert` variant	
	Performance	Cost	Performance	Cost	Performance	Cost	Performance	Cost
min-min	1.14 (±8%)	5,196 (±77%)	-	-	1.07 (±7%)	18,002 (±104%)	-	-
sufferage	1.17 (±15%)	5,494 (±80%)	-	-	1.06 (±10%)	21,984 (±99%)	-	-
static	2.29 (±34%)	2 (±107%)	-	-	1.56 (±28%)	4 (±91%)	-	-
static+max	2.57 (±45%)	3 (±106%)	-	-	1.82 (±37%)	4 (±91%)	-	-
static+readiness	2.19 (±33%)	3 (±104%)	1.48 (±24%)	7 (±92%)	1.53 (±29%)	4 (±92%)	1.19 (±13%)	13 (±65%)
static+readiness+max	2.53 (±44%)	3 (±101%)	1.50 (±25%)	7 (±87%)	1.81 (±36%)	4 (±88%)	1.22 (±15%)	14 (±65%)
dynamic1	2.41 (±38%)	10 (±80%)	1.34 (±17%)	14 (±79%)	1.67 (±40%)	11 (±77%)	1.10 (±9%)	19 (±72%)
dynamic1+max	2.84 (±45%)	14 (±85%)	1.43 (±23%)	18 (±74%)	2.09 (±45%)	16 (±86%)	1.17 (±13%)	26 (±73%)
dynamic2+010	2.58 (±41%)	42 (±117%)	2.03 (±41%)	21 (±108%)	1.66 (±39%)	43 (±112%)	1.44 (±31%)	28 (±93%)
dynamic2+010+max	3.61 (±53%)	46 (±104%)	3.32 (±54%)	38 (±108%)	2.09 (±45%)	47 (±99%)	1.92 (±38%)	50 (±103%)
dynamic2+100	2.99 (±56%)	8 (±104%)	2.83 (±49%)	10 (±87%)	1.72 (±38%)	9 (±88%)	1.69 (±39%)	19 (±66%)
dynamic2+100+max	4.26 (±60%)	10 (±78%)	4.65 (±51%)	17 (±87%)	2.14 (±45%)	12 (±72%)	2.09 (±33%)	29 (±94%)
randomtask	129.94 (±318%)	2 (±92%)	1.37 (±22%)	7 (±80%)	85.92 (±336%)	5 (±84%)	1.10 (±10%)	14 (±67%)

Table 3.3: Summary of the best heuristics according to their performance and their costs, for the three sizes of simulated platforms. Each value is an average of 48,000 configurations for 7 server platforms, of 2,400 configurations for 50 server platforms, and of 1,056 configurations for 100 server platforms.

	7 servers		50 servers		100 servers	
Heuristic	Performance	Cost	Performance	Cost	Performance	Cost
sufferage+insert	1.06 (±10%)	21,984 (±99%)				
min-min+insert	1.07 (±7%)	18,002 (±104%)				
dynamic1+mct+insert	1.10 (±9%)	19 (±72%)	1.05 (±9%)	873 (±117%)		
randomjob+mct+insert	1.10 (±10%)	14 (±67%)	1.05 (±8%)	1,356 (±204%)		
static+readiness+mct+insert	1.19 (±13%)	13 (±65%)	1.17 (±14%)	843 (±184%)		
dynamic1+mct	1.34 (±17%)	14 (±79%)	1.62 (±34%)	299 (±75%)	1.04 (±8%)	568 (±80%)
randomjob+mct	1.37 (±22%)	7 (±80%)	1.84 (±43%)	11 (±48%)	1.16 (±16%)	18 (±67%)
static+readiness+mct	1.48 (±24%)	7 (±92%)	1.88 (±40%)	10 (±53%)	1.19 (±21%)	17 (±63%)
static+readiness	2.19 (±33%)	3 (±104%)	4.37 (±61%)	1 (±41%)		
static	2.29 (±34%)	2 (±107%)				
randomjob	130 (±318%)	2 (±92%)	1,377 (±419%)	1 (±12%)		

One can imagine, in a first phase, to partition the application graphs and to assign one of the subsets defined by this partition to each of the servers. Vydyanathan, Khanna, Catalyurek, Kurc, Sadayappan, and Saltz [133] have presented such an approach, but their partitioning takes into account neither the heterogeneity of the platform nor the topology of the interconnection network.

There are two other problems with this work. The first one is that we have assumed unbounded memory on the data repositories, which may be an unrealistic assumption. The second one is that we made no hypotheses on the structure of the application graph: we studied the problem in a very general setting. Consequently, we were not able to do better than to design simple heuristics. The next study overcome both problems: we consider the special case of matrix multiplication and we assume that the memory of processors is bounded.

3.3 Revisiting matrix multiplication

Bibliographical note: The missing details and proofs can be found in [D9]. In particular, in this section we will discuss neither the complexity of the studied problem, nor the extension of the proposed approach to LU decomposition.

3.3.1 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 [33] and the ScaLAPACK outer product algorithm [24]. 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.

The computing platforms we target are composed of heterogeneous computing resources interconnected by a *sparse* network: there are no direct links between any pair of processors. Therefore, we are not interested in adapting the 2D processor grid strategy to heterogeneous clusters. Instead, we adopt an application scenario where input files are read from a fixed repository (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. This calls for a master-worker paradigm, or more precisely for a computational scheme where the master (the processor holding the input data) assigns computations to other resources, the workers. 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 our application scenario, it becomes necessary to include the cost of both the initial distribution of the matrices to the processors and of collecting back the 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[2] 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 volume of communication required 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? All of them, or maybe only the fastest computing ones, or else only the fastest-communicating ones? Once participating resources

[2]Of course, there are $\Theta(n^3)$ computations if we only consider algorithms that uses the standard way of multiplying matrices; this excludes Strassen's and Winograd's algorithms [48].

have been selected, there remain several scheduling decisions to take: 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 rest of this section is organized as follows. In Section 3.3.2, we state the scheduling problem precisely, and we introduce some notations. Next, in Section 3.3.3, 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. [130, 82] and we propose an algorithm almost achieving this bound on platforms with a single worker. We deal with homogeneous platforms in Section 3.3.4, and with heterogeneous ones in Section 3.3.5. We report on some MPI experiments in Section 3.3.6, and we conclude in Section 3.3.7.

3.3.2 Framework

Here, we formally state our hypotheses on the application and on the target platform.

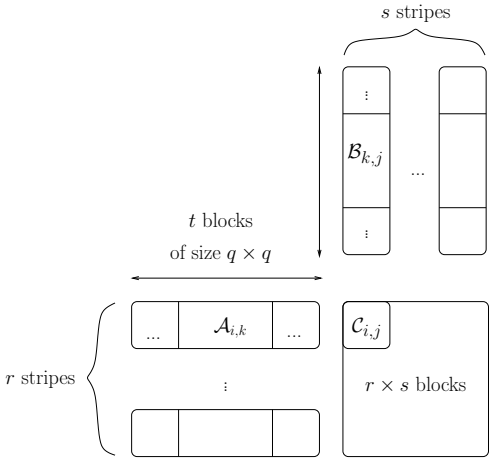


Figure 3.4: Partition of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

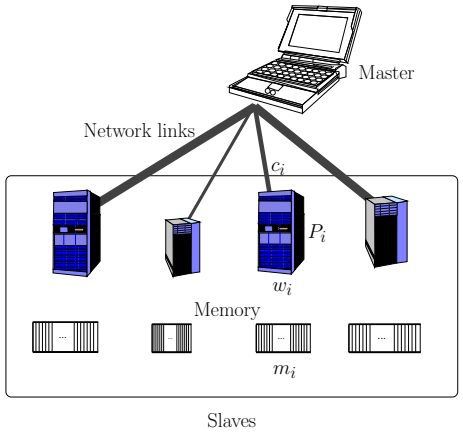


Figure 3.5: A fully heterogeneous master-worker platform.

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 3.4. More precisely:

- 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). This is to harness the power of Level 3 BLAS routines [23]. Typically, $q = 80$ or 100 when using ATLAS-generated routines [45, 134].

- 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 $t = n_{\mathcal{AB}}/q$.

- 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 each 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 same size. This will greatly simplify the analysis of communication costs.

Platform

We target a *star network* $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$, composed of a master P_0 and of p workers P_i, $1 \leq i \leq p$ (see Figure 3.5). 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:

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

Our star platforms are thus fully heterogeneous, both in terms of computations and of communications. A fully homogeneous star platform would be a star platform with identical workers and identical communication links: $w_i = w$ and $c_i = c$ for each worker P_i, $1 \leq i \leq p$. Without loss of generality, we assume that the master has no processing capability (otherwise, add a fictitious extra worker paying no communication cost to simulate computation at the master).

For the communication model, we once again use the strict *one-port* model, which is defined as follows:

- the master can only send data to, and receive data from, a single worker at a given time-step, and it cannot be enrolled in more than one communication at any time-step;
- a given worker cannot start execution before it has terminated the reception of the message from the master; similarly, it cannot start sending the results back to the master before finishing the computation.

Our final assumption is related to memory capacity; we assume that a worker P_i can only store m_i blocks (either from $\mathcal{A}$, $\mathcal{B}$, and/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.

3.3.3 Minimization of the communication volume

The complexity of our problem is open. We have shown [D9] that in the simple case with a single worker, without memory limits, and when $t = 1$, the obvious greedy algorithms are sub-optimal. We believe that the general problem is NP-complete but we were unable to prove it.

Our initial assumption is that communication costs dominate our problem. Therefore, we do not directly target makespan minimization, but communication volume minimization. Experiments, in Section 3.3.6, will show that our communication-volume minimization approach effectively leads to algorithms with shorter makespans.

We thus want to 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. As we are only interested in minimizing the total communication volume, we can simulate any parallel algorithm on a single worker and we only need to consider the one-worker case.

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 $\mathcal{C}_{ij}$, and
- 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 a lower bound on the communication volume established by Toledo et al. [130, 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.

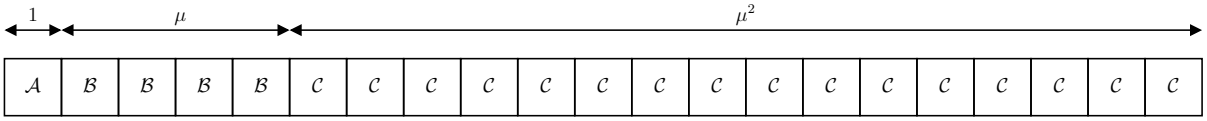


Figure 3.6: Memory layout for the *maximum re-use* algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.

Lower bound on the communication volume

To derive the lower bound, we refine an analysis due to Toledo [130]. The idea is to estimate the number of computations made thanks to m 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:

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

We then use Loomis-Whitney inequality [82]: in any algorithm that uses the standard way of multiplying matrices (this excludes Strassen's and Winograd's algorithm [48], 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 $K = \sqrt{N_A N_B N_C}$. Here $K = \sqrt{(\alpha_{old} + \alpha_{recv})(\beta_{old} + \beta_{recv})(\gamma_{old} + \gamma_{recv})}$. K is then maximized when $\alpha_{old} + \alpha_{recv} = \beta_{old} + \beta_{recv} = \gamma_{old} + \gamma_{recv} = \frac{2}{3}m$, and the lower bound for the communication-to-computation ratio is:

$$\text{CCR}_{\text{opt}} \geq \sqrt{\frac{27}{8m}}.$$

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

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 [130] 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 $\Omega(\frac{r^3}{\sqrt{m}})$ when his algorithm has a communication to computation ratio of $O(\frac{r^3}{\sqrt{m}})$. We can, however, still significantly improves the performance of matrix multiplication in our context, as our experiments will show in Section 3.3.6.

A closer look to 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 $\mathcal{B}$. Therefore, we designed an algorithm which reuses as much as possible the elements of $\mathcal{C}$.

Cannon's algorithm [33] and the ScaLAPACK outer product algorithm [24] both distribute square blocks of $\mathcal{C}$ to the processors. Intuitively, squares are better than elongated rectangles because their

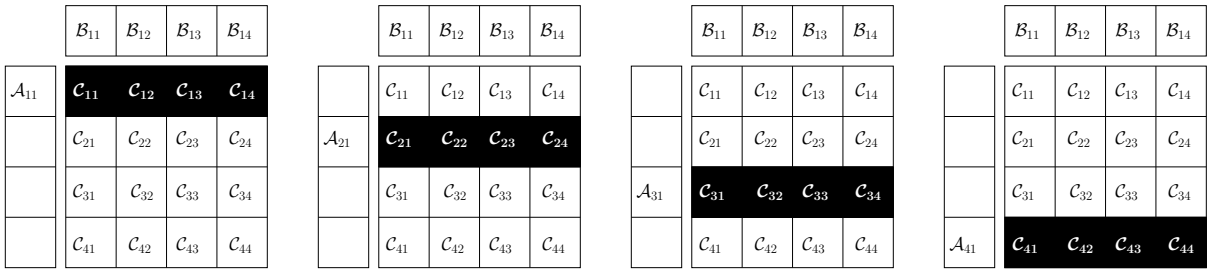


Figure 3.7: Four steps of the *maximum re-use* algorithm, with $m = 21$ and $\mu = 4$. The elements of $\mathcal{C}$ updated are displayed on white on black.

perimeter (which is proportional to the communication volume) is smaller for the same area. We use the same approach here, but we have not been able to assess any optimality result.

The *maximum re-use* algorithm uses the memory layout illustrated in Figure 3.6. Four consecutive execution steps are shown in Figure 3.7. Assume that there are m available buffers. First we find 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 3.6, 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:
Outer loop: while there remain $\mathcal{C}$ blocks to be computed

- Store μ^2 blocks of $\mathcal{C}$ in worker's memory: $\{\mathcal{C}_{i,j} \mid i_0 \leq i < i_0 + \mu,\ j_0 \leq j < j_0 + \mu\}$.
- **Inner loop**: For each k from 1 to t:
  1. Send a row of μ elements $\{\mathcal{B}_{k,j} \mid j_0 \leq j < j_0 + \mu\}$;
  2. Sequentially send μ elements of column $\{\mathcal{A}_{i,k} \mid i_0 \leq i < i_0 + \mu\}$. For each $A_{i,k}$, update μ elements of $\mathcal{C}$
- Return results to master.

The performance of one iteration of the outer loop of the *maximum re-use* algorithm can 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 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 [130].

Finally, we remark that the performance of the *maximum re-use* algorithm is quite close to the lower bound derived earlier:

$$\text{CCR}_\infty = \frac{2}{\sqrt{m}} = \sqrt{\frac{32}{8m}}.$$

3.3.4 Algorithms for homogeneous platforms

We now adapt the *maximum re-use* algorithm to fully homogeneous platforms. We have a limitation on the memory capacity. So we must first decide which part of the memory will be used to store which part of the original matrices, in order to maximize the total number of computations per time unit.

Principle of the algorithm

We load into the memory of each worker μ blocks of $\mathcal{A}$ and μ blocks of $\mathcal{B}$ to compute μ^2 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 μ^2 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 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

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

Indeed, this is the smallest value to saturate the communication capacity of the master required to sustain the corresponding computations. Finally, we cannot use more processors than are available and, 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 elementary communication and computation times. Hence we 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 that s is divisible 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 decomposed into two parts. Algorithm 3 outlines the program of the master, while Algorithm 4 is the program of each worker.

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 decided to 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, 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 lost is at most 4%, which is small enough to be neglected. Note that it would be technically 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 become much more complicated.

3.3.5 Algorithms for heterogeneous platforms

We now consider the general problem, i.e., when processors are heterogeneous in term of memory size as well as computation or communication time. 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 the memory changes from worker to worker, because it depends upon their memory capacities. We first compute all the different values of μ_i, μ_i being the largest integer such that:

$$\mu_i^2 + 4\mu_i \leq m_i.$$

Algorithm 3: Homogeneous version, master program.

$\mu \leftarrow \left\lfloor \sqrt{4+m} - 2 \right\rfloor$
$\mathfrak{P} \leftarrow \min\left\{p, \left\lceil \frac{\mu w}{2c} \right\rceil\right\}$
Split the matrix into squares $\mathbf{C}_{i',j'}$ of μ^2 blocks (of size $q \times q$):
$\mathbf{C}_{i',j'} = \{\mathcal{C}_{i,j} \mid (i'-1)\mu + 1 \leq i \leq i'\mu, (j'-1)\mu + 1 \leq j \leq j'\mu\}$
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 4: 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

To adapt our *maximum re-use* algorithm to heterogeneous platforms, we first present a steady-state-like approach and discuss its limitations. We then introduce our final algorithm for heterogeneous platforms.

Bandwidth-centric resource selection

Each worker P_i has parameters c_i, w_i, and μ_i, and each participating P_i needs to receive $2\mu_i t c_i$ blocks to perform $t\mu_i^2 w_i$ computations. Once again, we neglect I/O for $\mathcal{C}$ blocks. 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 communication —the master has limited bandwidth— and of computation —a worker has limited computing power and cannot perform more work than it receives. The objective is to maximize the amount of work performed per time-unit. Altogether, we gather the linear program presented on Figure 3.4.

The optimal solution for this system is a bandwidth-centric strategy [10, 6]: 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 3.5. Using the bandwidth-centric

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

Table 3.4: Linear program maximizing the amount of work performed per time-unit.

	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 3.5: Platform for which the bandwidth centric solution is not feasible.

strategy, every 160 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 $\sqrt{2x}$ blocks and as many buffers. As x can be arbitrary large, 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 can be seen as an upper bound of the performance that can be achieved.

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, while simplifying 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.

There remains to decide which processor to select at each step of the first phase. 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 for this algorithm 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.

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. Instead, 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.

3.3.6 MPI experiments

In this subsection, 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 target heterogeneous platforms, and we assess the impact of the degree of heterogeneity (in processor speed, link bandwidth, and memory capacity) on the performance of the various algorithms.

Platforms

We used a heterogeneous cluster composed of twenty-seven processors located in Lyon. It is composed of four different homogeneous sets of machines. The different sets are composed of: 1) 8 SuperMicro servers 5013-GM, with processors P4 2.4 GHz; 2) 5 SuperMicro servers 6013PI, with processors P4 Xeon 2.4 GHz; 3) 7 SuperMicro servers 5013SI, with processors P4 Xeon 2.6 GHz; 4) 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. As this platform may not be as heterogeneous as we would like, we sometimes artificially modify its heterogeneity. In order to artificially slow down a communication link, we send several times the same message 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. In all experiments, except the last batch, we used nine processors: one master and eight workers.

Algorithms

We choose four different algorithms from the literature which we compare our algorithms to. The closest work addressing our problem is Toledo's out-of-core algorithm [130]. Hence, this work will serve as the baseline reference. Then we will study hybrid algorithms, i.e., algorithms which use our memory layout and are based on classical principles such as round-robin, min-min [97], or a dynamic demand-driven approach. The first six algorithms below use our memory allocation, the only difference between them is the order in which the master sends blocks to workers.

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 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 which 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 considered processor speed, and so on). We do this process for all the existing values, and we pick the virtual platform which minimizes the total execution time.

Heterogeneous algorithm (**Het**) is our heterogeneous algorithm. As we can have eight different versions of the resource selection, in a first step we simulate the eight versions, and then we pick up and run the best one.

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.

Block Matrix Multiply (**BMM**) is Toledo's algorithm [130]. 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 $\mathcal{C}$, the square blocks having the same size. It sends blocks to the workers in a demand-driven fashion.

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.

Note that the six 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 $\mathcal{C}$.

When launching an algorithm on the platform, the very first step we do is to determine the platform's parameters. In 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 20 and 80 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 2% 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.

Experimental results

In the first three sets of experiments, we only have one parameter of heterogeneity, either the amount of memory, or the communication speed, or the computation speed. We test the algorithms on such platforms with five matrices of increasing sizes. 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). 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.

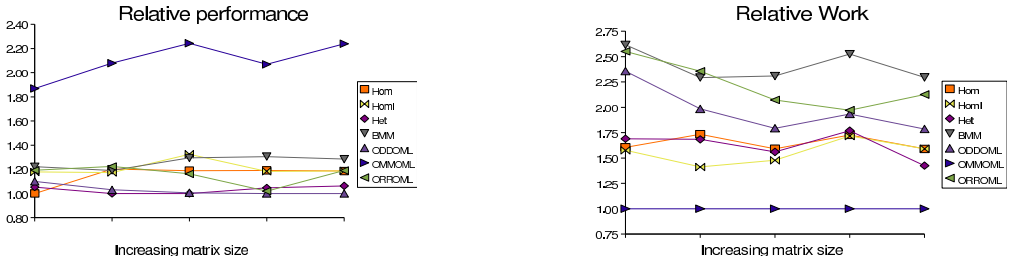
As we want to assess whether the performance of any studied algorithm depends on the matrix size, we look at the *relative performance* of the algorithms rather that at their absolute execution times. The relative performance of a given algorithm on a particular instance is equal to the makespan achieved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

Beside relative performance, we take into account the number of processors used. To assess the

efficiency of a given algorithm, we look at its *relative work*, which is equal, for a given instance, to its makespan times the number of enrolled processors, divided by the minimum of this value over all studied algorithms.

Heterogeneous memory size

Here we assess the performance 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.



(a) Performance of algorithms.

(b) Efficiency of algorithms.

Figure 3.8: Heterogeneous memory.

Figure 3.8(a) presents the relative performance 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 execution times, **Het** needs about 2000 seconds to compute the product of the smallest matrices, and about 3500 seconds for the largest.

The variations in the performance 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 the 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 performance, they do not prevent it to sometimes achieve very bad performance.

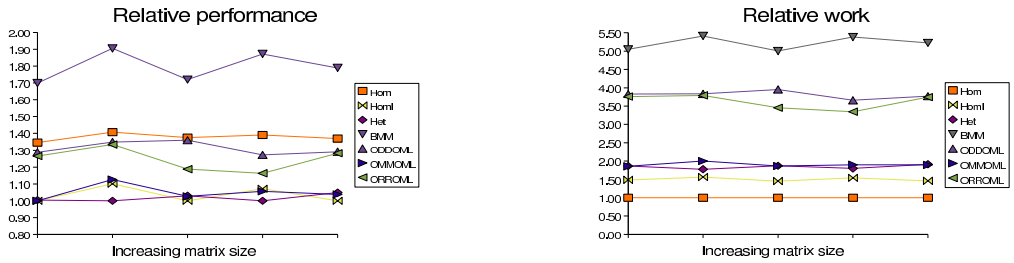
The ranking of the algorithms is quite different when we look at the relative work (Figure 3.8(b)). First we have **OMMOML** which only uses two workers, and is thus very thrifty, at the expense of its absolute performance. Then we have **HomI**, **Het**, **Hom**, and **ODDOML**. **Hom** relative performance is always better than that of **ODDOML**: **Hom** is performing some resource selection contrarily to **ODDOML** which always uses all the processors. And **HomI** is even better than **Hom**, thanks to a better platform selection. This is also the reason why the gap between the relative work of **Het** and that of **ODDOML** is significantly larger than the gap between their relative performances. Finally, we have **ORROML** and **BMM**, which do not achieve an efficient makespan and make no resource selection, and thus achieve a very bad relative work.

Heterogeneous communication links

We now assess the performance 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 3.9(a) shows the relative performance. 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 on Figure 3.9(b)). The first figure also shows the gap between **HomI** and **Hom**: **Hom** performs close



(a) Performance of algorithms.

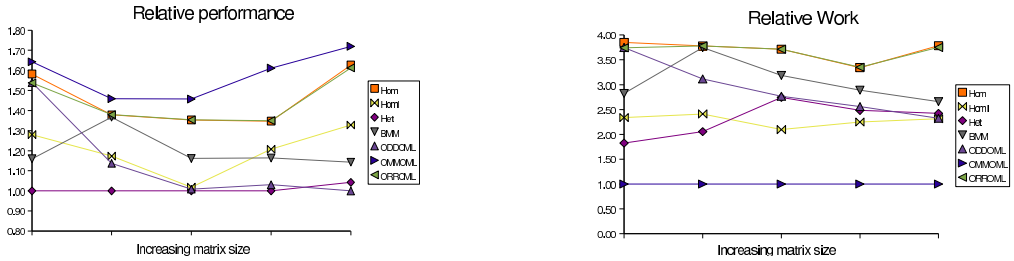
(b) Efficiency of algorithms.

Figure 3.9: Heterogeneous communication links.

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, and makes no resource selection, which explains its worse relative work. **BMM** achieves a makespan which is 70 to 90 percent worse than the best one.

Heterogeneous computations

Here we assess the performance 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 $S/4$.



(a) Performance of algorithms.

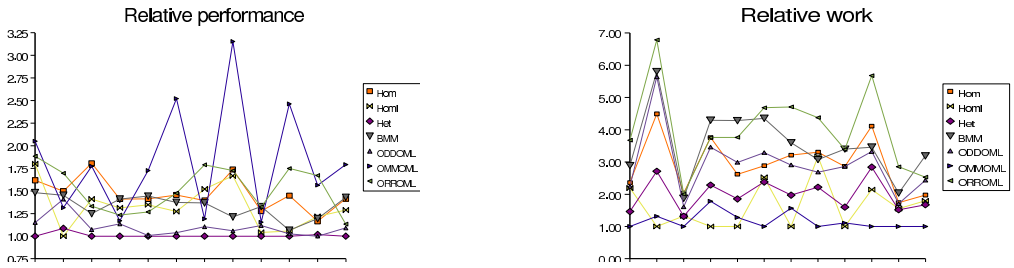
(b) Efficiency of algorithms.

Figure 3.10: Heterogeneous computations.

In Figure 3.10(a), we see the relative performance obtained during this set of experiments. **BMM** performs rather well, but its makespan is larger than that of **Het**. Moreover, looking at the relative work in Figure 3.10(b), we see that the gap between the algorithms becomes larger, as our algorithms enroll fewer resources during execution. Among the other algorithms, we see that **ODDOML** performs well. If we look at the relative performance, we also see that **Het** uses more and more processors as matrix size increases.

Fully heterogeneous platforms

We now consider fully heterogeneous platforms. 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 either 2 in the first setting or 4 in the second one (first two columns on Figures 3.11(a) and 3.11(b)). In order to show that our heterogeneous algorithm works on any heterogeneous platform, we also randomly create ten different platforms (last ten columns on 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.



(a) Performance of algorithms.

(b) Efficiency of algorithms.

Figure 3.11: Fully heterogeneous platforms.

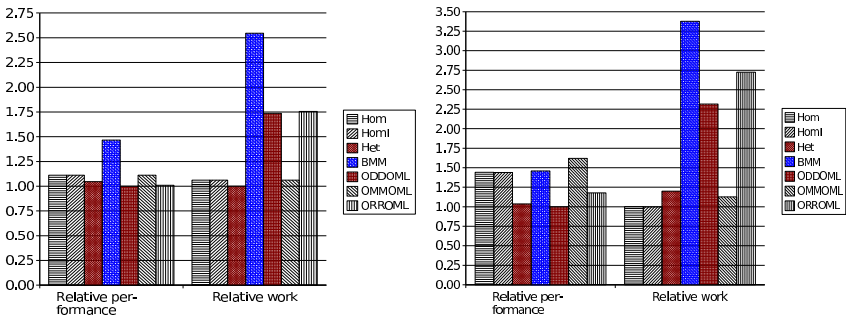
The results of these experiments are summarized on Figures 3.11(a) and 3.11(b). We see that **Het** achieves the best makespan for all but two of the 12 platforms, and in the remaining cases is no more than 9% and 2% away from the best studied algorithm. All the other algorithms are, at least once, more 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 makespans on average but, as it does not select resources, its relative work is far worse. The relative work of **Het** is the best among all algorithms except **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 resource selection, the makespan it achieves can be up to 80% larger than the best makespan, and is 34% larger on average.

Real platform

In this set of experiments, we use almost all the processors of our platform. We do not modify the communication speed nor the computation speed 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 3.12(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 3.12(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 $\mathcal{B}$.



(a) August 2007 configuration

(b) November 2006 configuration

Figure 3.12: Real platform.

The results of these experiments are summarized on Figure 3.12. The result on the actual platform are similar to those obtained on homogeneous platforms [D9]. All the algorithms but **BMM** have similar

makespan. All algorithms making resource selection use eleven worker among the twenty available, which explains why they achieve similar relative work.

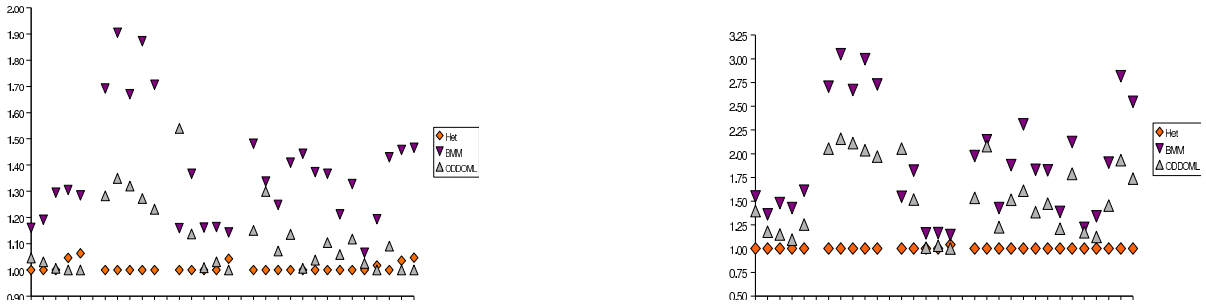
The results of the experiments on the older version of the platform are very similar on the ones previously obtained on memory heterogeneous platforms. **ODDOML** and our heterogeneous algorithm **Het** achieve the best *makespans*. Then we have **ORROML**, our homogeneous algorithms, **BMM**, and finally **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 one which uses the whole platform. On another side, **Hom**, **HomI**, **OMMOML** use six workers with small memories. All other algorithms use the whole platform. We can thus see the impact of the resource selection on the relative work of the algorithms.

Summary

Figure 3.13 summarizes all our MPI experiments. Figures 3.13(a) and 3.13(b) respectively present the relative performance and the relative work obtained for each experiment by our heterogeneous algorithm (**Het**), by Toledo's algorithm (**BMM**), and by the best of the dynamic heuristics using our memory layout (**ODDOML**). The results show the superiority of our memory allocation. Furthermore, if we add the resource selection of **Het**, not only do we achieve, most of the time, the best makespan, but also the best relative work as we also spare resources. Using our memory layout (**ODDOML**) rather than Toledo's (**BMM**) enables us to gain 19% of execution time on average. When this is combined with resource selection, this enables us to gain additionally 10%, that is 27% against Toledo's running time. We achieve this significant gain while sparing resources. Our **Het** algorithm is on average 1% away from the best achieved makespan. At worst **Het** is 14% away from the best makespan, **ODDOML** 61%, and **BMM** 128%. Moreover, we have seen that 80% of the time, the performance of **Het** was in fact obtained thanks to a global resource selection.

The steady-state approach described in Section 3.3.5 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 performance 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.



(a) Relative performance.

(b) Relative work.

Figure 3.13: Summary of experiments.

3.3.7 Conclusion

In this study, we have been able to design an efficient matrix-multiplication algorithm targeting heterogeneous master-worker platforms. This algorithm is built on a communication minimization principle, which itself is derived from a careful optimization of the use of memory.

Contrarily to the two previous studies in this chapter, we had a precise knowledge of the dependences of atomic computations on atomic data. The relation dependences were also regular. This knowledge

enabled us to derive a static algorithm significantly more efficient and more thrifty than classical dynamic heuristics.

Chapter 4

Online scheduling of divisible requests

Bibliographical note: The missing details and proofs can be found in [D1, D11].

In the previous chapter we have mainly focused on the importance and impact of communications. Ironically, in the current chapter, we are just going to forget about communications! As we have already stressed, the models used must be accurate. It happens that the communications involved in the application studied in this chapter are actually negligible. So, we neglect them.

The solutions presented in the two previous chapters were mostly *static* solutions to *static* problems: all the problem characteristics were supposed to be known in advance and we were thus able to derive a solution beforehand and once and for all. In this chapter we are considering an online system, that is a system which receives requests over time, from several users, without any a priori knowledge. We have thus to design some dynamic solutions, that is solutions able to deal with the problem dynamicity. In order to build those solutions on solid foundations and to be able to assess their quality against absolute references, we start by studying the theoretical version of our problem, i.e., the offline version, where everything is supposed to be known in advance but where the different jobs still have different release dates. We will see that we are able to design very efficient online solutions. An amusing conclusion of the study being that the heuristics which may be the more intellectually satisfying, the guaranteed ones, turned out to be bad choices. When targeting actual applications, one must always look for heuristics which are efficient in *practice* !

4.1 Introduction

The problem of searching large-scale genomic and proteomic sequence databanks is an increasingly important bioinformatics problem. In this chapter we study the deployment of such applications on heterogeneous parallel computing environments. In fact, these applications are part of a larger class of applications, in which each task in the application workload exhibits an "affinity" for particular nodes of the targeted computational platform. In the genomic sequence comparison scenario, the presence of the required databank on a particular node is the sole factor that constrains task placement decisions. In this context, task affinities are determined by location and replication of the sequence databanks in the distributed platform.

Numerous efforts to parallelize biological sequence comparison applications have been realized (e.g., [29, 50, 101]). These efforts are facilitated by the fact that such biological sequence comparison algorithms are typically computationally intensive, embarrassingly parallel workloads. In the scheduling literature, this computational model is effectively a *divisible workload scheduling* problem with negligible communication overheads. The work presented in this chapter concerns this application model in the context of *online scheduling* (i.e., in which the scheduler has no knowledge of any job in the workload in advance of its release date). Thus far, this specific problem has not been considered in the scheduling literature.

Aside from divisibility, the main difference with classical scheduling problems lies in the fact that the platforms we target are shared by many users. Consequently, we need to ensure a certain degree of fairness between the different users and requests. Defining a fair objective that accounts for the various

job characteristics (release date, processing time) is thus the first difficulty to overcome. After having presented our motivating application and our framework in Section 4.2, we explain in Section 4.3 why we choose as our objective the minimization of the stretch of jobs. To have a good background on related objectives functions and results, in Section 4.4 we focus on the max-flow and sum-flow metrics. Then in Section 4.5 we study sum-stretch optimization, in Section 4.6 offline max-stretch optimization, and in Section 4.7 Pareto offline optimization of max-stretch. Building on the previous sections, we focus in Section 4.8 on the online optimization of max-stretch. In Section 4.9 we summarize the known and new results on complexity. Finally, we present in Section 4.10 an experimental evaluation of the different solutions proposed, and we conclude in Section 4.11.

4.2 Motivating application and framework

Motivating application

The only purpose of this section is to present the application that originally motivated this work, the GriPPS [25, 68] protein comparison application. The GriPPS framework is based on large databases of information about proteins; each protein is represented by a string of characters denoting the sequence of amino acids of which it is composed. Biologists need to search such sequence databases for specific patterns that indicate biologically significant structures. The GriPPS software enables such queries in grid environments, where the data may be replicated across a distributed heterogeneous computing platform. To develop a suitable application model for the GriPPS application scenario, we performed a series of experiments to analyze the fundamental properties of the sequence comparison algorithms used in this code. Here we only report on the conclusions of this study (whose details can be found in [D11]).

From our modeling perspective, the critical components of this application are:

1. **Protein databanks**: the reference databases of amino acid sequences, located at fixed locations in a distributed heterogeneous computing platform.
2. **Motifs**: compact representations of amino acid patterns that are biologically important and serve as user input to the application.
3. **Sequence comparison servers**: computational processes co-located with protein databanks that accept as input sets of motifs and return as output all matching entries in any subset of a particular databank.

The main characteristics of the GriPPS application are:

1. **Negligible communication costs**. A motif is a relatively compact representation of an amino acid pattern. Therefore, the communication overhead induced while sending a motif to any processor is negligible compared to the processing time of a comparison.
2. **Divisible loads**. The processing time required for comparing a sequence against a subset of a particular databank is linearly proportional to the size of the subset relative to the entire databank. This property allows us to distribute the processing of a request among many processors at the same time without additional cost.

   The GriPPS protein databank search application is therefore an example of a *linear divisible workload without communications*.

   In the classical scheduling literature, preemption is defined as the ability to suspend a job at any time and to resume it, possibly on another processor, at no cost. Our application implicitly falls in this category. Indeed, we can easily halt the processing of a request on a given processor and continue the pattern matching for the unprocessed part of the database on a different processor (as it only requires a negligible data transfer operation to move the pattern to the new location). From a theoretical perspective, divisible load without communications can be seen as a generalization of the *preemptive execution model* that allows for simultaneous execution of different parts of a same job on different machines.
3. **Uniform machines with restricted availabilities**. A set of jobs is uniform over a set of processors if the relative execution times of jobs over the set of processors does not depend on the

nature of the jobs. More formally, for any job J_j, $p_{i,j}/p_{i',j} = k_{i,i'}$, where $p_{i,j}$ is the time needed to process job J_j on processor i. In essence, $k_{i,i'}$ describes the relative power of processors i and i', regardless of the size or the nature of the job being considered. Our experiments indicated a clear constant relationship between the computation time observed for a particular motif on a given machine, compared to the computation time measured on a reference machine for that same motif. This trend supports the hypothesis of uniformity. However, in practice a given databank may not be available on all sequence comparison servers. Our model essentially represents a *uniform machines with restricted availabilities* scheduling problem, which is a specific instance of the more general *unrelated machines* scheduling problem.

Framework and notations

Formally, an instance of our problem is defined by n jobs, J_1, ..., J_n and p machines (or processors), M_1, ..., M_p. The job J_j arrives in the system at time r_j (expressed in seconds), which is its release date; we suppose that jobs are numbered by increasing release dates. The time at which job J_j is completed is denoted as C_j. Then, the *flow time* of the job J_j, defined as $F_j = C_j - r_j$, is essentially the time the job spends in the system. The value $p_{i,j}$ denotes the amount of time it would take for machine M_i to process job J_j. Note that $p_{i,j}$ can be infinite if the job J_j cannot be executed on the machine M_i, e.g., for our motivating application, if job J_j requires a databank that is not present on the machine M_i. Finally, each job is assigned a *weight* or *priority* w_j.

Due to the divisible load model, each job may be divided into an arbitrary number of sub-jobs, of any size. Furthermore, each sub-job may be executed on any machine at which the data dependences of the job are satisfied. Thus, at a given moment, many different machines may be processing the same job (with a master scheduler ensuring that these machines are working on *different* parts of the job). Therefore, if we denote by $\alpha_{i,j}$ the fraction of job J_j processed on M_i, we enforce the following property to ensure each job is fully executed: $\forall j, \sum_i \alpha_{i,j} = 1$.

When a size W_j can be defined for each job J_j —e.g., in the uni-processor case— we denote by Δ the ratio of the sizes of the largest and shortest jobs submitted to the system: $\Delta = \frac{\max_j W_j}{\min_j W_j}$.

As we have seen, for the particular case of our motivating application, we could replace the unrelated times $p_{i,j}$ by the expression $W_j \cdot \mathcal{C}_i$, where W_j denotes the size (in Mflop) of the job J_j and $\mathcal{C}_i$ denotes the computational capacity of machine M_i (in second·Mflop^{-1}). To maintain correctness for the biological sequence comparison application, we separately maintain a list of databanks present at each machine and enforce the constraint that a job J_j may only be executed on a machine that has a copy of all data upon which job J_j depends. However, since the theoretical results we present do not rely on these restrictions, we retain the more general scheduling problem formulation (i.e., unrelated machines). As a consequence, all the values we consider in this chapter are nonnegative rational numbers (except the previously mentioned case in which $p_{i,j}$ is infinite if J_j cannot be processed on M_i).

Relationships with the uni-processor case with preemption

We first state that any schedule in the uniform machines model with divisibility has a canonical corresponding schedule in the uni-processor model with preemption. This is especially important as many interesting results in the scheduling literature only hold for the preemptive computation model (denoted *pmtn*).

Lemma 1. *For any platform M_1, ..., M_p composed of uniform processors, i.e., such that for any job J_j, $p_{i,j} = W_j \cdot wi$, one can define a platform made of a single processor $\widetilde{M}$ with $\widetilde{p} = 1/\sum_i \frac{1}{wi}$, such that:*

For any divisible schedule $\mathcal{S}$ of J_1, ..., J_n on $\{M_1, ..., M_p\}$ there exists a preemptive schedule $\widetilde{\mathcal{S}}$ of J_1, ..., J_n on $\widetilde{M}$ such that no job has a greater completion time under $\widetilde{\mathcal{S}}$ than under $\mathcal{S}$.

Figure 4.1 illustrates how the schedule $\widetilde{\mathcal{S}}$ can be derived from $\mathcal{S}$.

As a consequence of the above lemma, any complexity result for the preemptive uni-processor model also holds for the uniform divisible model. Thus, throughout this chapter, in addition to addressing the multi-processor case, we will also closely examine the uni-processor case.

Unfortunately, this line of reasoning is no longer valid when the computational platform exhibits restricted availabilities. In the uni-processor case, a schedule can be seen as a priority list of the jobs (see the article of Bender, Muthukrishnan, and Rajaraman [17] for example). For this reason, whenever

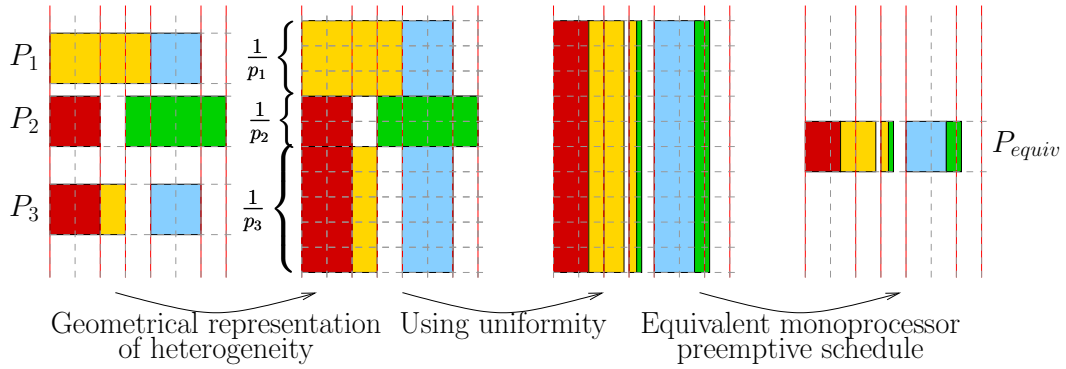


Figure 4.1: Geometrical transformation of a divisible uniform problem into a preemptive uni-processor problem

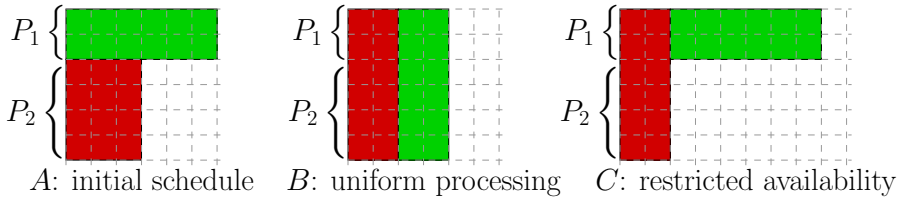


Figure 4.2: Illustrating the difference between the uniform model and the restricted availability model.

we will present heuristics for the uniprocessor case they will follow the same basic approach: maintain a priority list of the jobs and at any moment, execute the one with the highest priority. In the multi-processor case with restricted availability, an additional scheduling dimension must be resolved: the spatial distribution of each job.

The example in Figure 4.2 explains the difficulty of this last problem. In the uniform situation, it is always beneficial to fully distribute work across all available resources: each job's completion time in situation B is strictly better than the corresponding job's completion time in situation A. However, introducing restricted availability confounds this process. Consider a case in which tasks may be limited in their ability to utilize some subset of the platform's resources (e.g., their requisite data are not present throughout the platform). In situation C of Figure 4.2, one task is subject to restricted availability: the P_2 computational resource is not able to service this task. When restricted availabilities are introduced (situation C), the completion time vectors can no longer be compared, and we no longer have a schedule which dominate all others. For this reason our problem is related to the preemptive uni-processor model, but is not equivalent to it.

In order to reuse the existing results, we apply the following simple rule to extend to general platforms the heuristics designed for uni-processor systems:

1 **while** *some processors are idle* **do**
2 Select the job with the highest priority and distribute its processing on all appropriate processors that are available.

4.3 The objective: stretch minimization

As we are targeting an online system where jobs are submitted over time, we need to use an objective function considering the job release dates. Another important characteristic of our problem is that we target a platform shared by many users. As a consequence, we need to ensure a certain degree of fairness between the different requests. Given a set of requests, how should we share resources amongst the different requests? Following the discussion of Section 1.3, an appealing objective is then the minimization of the weighted flow or of one of its variant, the stretch. The following theorem states that

the simultaneous optimization of the max-stretch and sum-stretch is impossible in certain situations.

Theorem 10. *Consider any online algorithm which has a competitive ratio of $\rho(\Delta)$ for the sum-stretch. We assume that this competitive ratio is not trivial, i.e., that $\rho(\Delta) < \Delta^2$. Then, there exists for this algorithm a sequence of jobs that leads to starvation, and thus for which the obtained max-stretch is arbitrarily greater than the optimal max-stretch.*

Using the exact same construction, we can show that for any online algorithm which has a non-trivial competitive ratio of $\rho(\Delta) < \Delta$ for the sum-flow, there exists a sequence of jobs leading to starvation and where the obtained max-flow is arbitrarily greater than the optimal one.

We must comment on our assumption about *non-trivial competitive ratios*. This comes from the fact that ignoring job sizes leads, on a single processor, to a Δ^2-competitive online algorithm for sum-stretch and Δ-competitive online algorithm for sum-flow:

Theorem 11. First come, first served *is:*

- *Δ^2-competitive for the online minimization of sum-stretch,*
- *Δ-competitive for the online minimization of max-stretch,*
- *Δ-competitive for the online minimization of sum-flow, and*
- *optimal for the online minimization of max-flow (classical result, see Bender et al. [15] for example).*

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 (in our context the weighted flow or the stretch of each job) is made as close to the other ones as possible. Sum-based metrics tend to optimize instead the *utilization* of the platform. The previous theorem establishes that these two objectives can be in opposition on particular instances. As a consequence, it should be noted that any algorithm optimizing a sum-based metric has the particularly undesirable property of potential starvation. This observation, coupled with the fact that the stretch is more relevant than the flow in a system with highly variable job sizes, motivates max-stretch as the metric of choice in designing scheduling algorithms in our setting. As the minimization of sum-stretch was far more studied in the literature than that of the max-stretch, we also discuss sum-stretch minimization.

4.4 Flow optimization

On a single processor, the max-flow is optimized by FCFS (see Bender *et al.* [15] for example). Using the remarks of Section 4.2, we can easily derive an optimal online algorithm for $\langle Q|r_j; div|F_{\max}\rangle$. We will see in Section 4.6 that $\langle R|r_j; div|\max w_j F_j\rangle$ can be solved in polynomial time using linear programing techniques; this is therefore also true for $\langle R|r_j; div|F_{\max}\rangle$.

Regarding, sum-flow, it was proved by Baker [4], using exchange arguments, that SRPT (shortest remaining processing time first) is optimal for the $\langle 1|r_j; pmtn|\sum C_j\rangle$ problem. It is thus also optimal for $\langle 1|r_j; pmtn|\sum F_j\rangle$ and, using the remarks of Section 4.2, we can easily derive an online optimal algorithm for $\langle Q|r_j; div|\sum F_j\rangle$. Under the *uniform machines with restricted availabilities* model, however, this problem becomes NP-complete. As a consequence:

Theorem 12. *The scheduling problem $\langle R|r_j, div|\sum F_j\rangle$ is NP-complete.*

4.5 Sum-stretch optimization

We first establish the complexity of sum-stretch minimization in our framework. Then, we focus on the one processor setting and study the competitiveness of "classical" heuristics. We see that even if sum-stretch minimization looks to be a hard problem, some simple heuristics exhibit good performance.

Complexity of the offline problem. In the general case, without preemption and divisibility, minimizing the sum-stretch is an NP-complete problem:

Theorem 13. *The scheduling problem* $\langle 1|r_j|\sum S_j\rangle$ *is NP-complete.*

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. In the framework with preemption, Bender, Muthukrishnan, and Rajaraman [17] present a Polynomial Time Approximation Scheme (PTAS) for sum-stretch minimization. Chekuri and Khanna [42] present an approximation scheme for the more general sum weighted flow minimization problem. As these approximation schemes cannot be extended to work in an online setting, we will not discuss them further.

Moving to the divisible load framework, we can easily say that the complexity of $\langle Q|r_j; div|\sum S_j\rangle$ is open (using the remarks of Section 4.2). The minimization of the sum-stretch is however NP-complete on unrelated machines:

Theorem 14. *The scheduling problem* $\langle R|r_j, div|\sum S_j\rangle$ *is NP-complete.*

Lower bound on the competitiveness of online algorithms. Muthukrishnan, Rajaraman, Shaheen, and Gehrke [103] propose an optimal online algorithm when there are only two job sizes. Mainly, they prove that there is no optimal online algorithm for the sum-stretch minimization problem when there are three or more distinct job sizes. Furthermore, they give a lower bound of 1.036 on the competitive ratio of any online algorithm. This bound can be slightly improved:

Theorem 15. *No online algorithm minimizing the sum-stretch with preemption has a competitive ratio less than or equal to 1.19484.*

The fact that this bound is so low is in part explained by the performance of SRPT for sum-stretch.

Shortest remaining processing time (SRPT). In the previous section, we have recalled that *shortest remaining processing time* (SRPT) is optimal for minimizing the sum-flow. When SRPT takes a scheduling decision, it only considers the *remaining* processing time of a job, and not its *original* processing time. Therefore, from the point of view of the sum-stretch minimization, SRPT does not take into account the *weight* of the jobs in the objective function. Nevertheless, Muthukrishnan, Rajaraman, Shaheen, and Gehrke have shown [103] that SRPT is 2-competitive for sum-stretch.

Smith's ratio rule. Another well studied algorithm is the Smith's ratio rule [125] also known as *shortest weighted processing time* (SWPT). This is a preemptive list scheduling where the available jobs are executed in increasing value of the ratio $\frac{p_j}{w_j}$. Whatever the weights, SWPT is 2-competitive [117] for the minimization of the sum of weighted completion times ($\sum w_j C_j$). Note that a ρ-competitive algorithm for the sum weighted flow minimization ($\sum w_j(C_j - r_j)$) is ρ-competitive for the sum weighted completion time ($\sum w_j C_j$). However, the reverse is not true: a guarantee on the sum weighted completion time ($\sum w_j C_j$) does not induce any guarantee on the sum weighted flow ($\sum w_j(C_j - r_j)$). Therefore, the previous ratio on the minimization of the sum of weighted completion times gives us no result on the efficiency of SWPT for the minimization of the sum-stretch. Furthermore, we can even prove that SWPT is not a competitive algorithm for minimizing the sum-stretch. Indeed, SWPT schedules the available jobs by increasing values of $\frac{1}{p_j^2}$ and has thus exactly the same behavior as the *shortest processing time* first heuristic (SPT). The following theorem states that SPT (and thus SWPT) is not a competitive algorithm for minimizing the sum-stretch.

Theorem 16. *For any value* $\rho > 1$*, there is an instance on which the sum-stretch realized by* SPT *is at least* ρ *times the optimal. Furthermore, we can impose that in this instance* Δ*, the ratio of the sizes of the largest and shortest jobs submitted to the system, is equal to 2.*

Shortest weighted remaining processing time (SWRPT). The weakness of the SWPT heuristics is obviously that it does not take into account the remaining processing times: it may preempt a job when it is almost completed. 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. This

is what the *shortest weighted remaining processing time* heuristic (SWRPT) does. In the framework of sum-stretch minimization, at any time t, SWRPT schedules the job J_j which minimizes $p_j\rho_t(j)$. Muthukrishnan, Rajaraman, Shaheen, and Gehrke [103] proved that SWRPT is actually optimal when there are only two job sizes.

Neither of the proofs of competitiveness of SRPT or SWPT can be extended to SWRPT. SWRPT has apparently been studied by Megow [99], but only in the scope of the sum weighted completion time. So far, there is no guarantee on the efficiency of SWRPT for sum-stretch minimization. Intuitively, we would think that SWRPT is more efficient than SRPT for the sum-stretch minimization. However, the following theorem shows that the worst case for SWRPT for the sum-stretch minimization is no better than that of SRPT.

Theorem 17. *For any real ϵ, $1 > \epsilon > 0$, there exists an instance such that* SWRPT *is not $(2-\epsilon)$-competitive for the minimization of the sum-stretch.*

4.6 Offline max-stretch optimization

Bender, Chakrabarti, and Muthukrishnan [15] have shown that the problem of max-stretch minimization on one machine *without* preemption, i.e., problem $\langle 1|r_j|S_{\max}\rangle$, cannot be approximated within a factor $O(n^{1-\epsilon})$ for arbitrarily small $\epsilon > 0$, unless P=NP. In this section, we show that if we allow the divisibility of loads, we are able to minimize the maximum weighted flow in polynomial time even on unrelated machines. (In fact, as we showed in [D1, C12], this is also true if we allow preemption.)

It should be noted that, prior to our work, at least two solutions were known for minimizing the max-stretch on one machine with preemption. Baker, Lawler, Lenstra, and Rinnooy Kan [5] presented an $O(n^2)$ algorithm to solve an even more general problem: $\langle 1|pmtn, prec, r_j|f_{\max}\rangle$ (where $f_{\max}$ is the maximum of the costs of the jobs and the cost of a job is a non-decreasing function of its completion time). This algorithm determines the job of least priority and then iterates. Another solution using network flow maximization techniques was known (We do not know any reference to this technique who was presented to us by Michael Bender); we recall it in [D1]. In our divisible load framework, we do not know how to extend this flow maximization technique to solve the case of uniform machines with restricted availabilities, much less the more general case of unrelated processors.

We began the study of our problem by stating the relationship between deadline scheduling and minimization of the maximum weighted flow.

Max weighted flow minimization and deadline scheduling

Let us assume that we are looking for a schedule $\mathcal{S}$ under which the maximum weighted flow is less than or equal to some objective value $\mathcal{F}$. The weighted flow of any job J_j is equal to $w_j(C_j - r_j)$. Then, due to our hypothesis on $\mathcal{F}$, we have:

$$\max_{1\leq j\leq n} w_j(C_j - r_j) \leq \mathcal{F} \quad \Leftrightarrow \quad \forall j \in [1;n],\ w_j(C_j - r_j) \leq \mathcal{F} \quad \Leftrightarrow \quad \forall j \in [1;n],\ C_j \leq r_j + \mathcal{F}/w_j.$$

Thus, the execution of J_j must be completed before time $\bar{d}_j(\mathcal{F}) = r_j + \mathcal{F}/w_j$ for schedule $\mathcal{S}$ to satisfy the bound $\mathcal{F}$ on the maximum weighted flow. Therefore, looking for a schedule which satisfies a given upper bound on the maximum weighted flow is equivalent to an instance of the deadline scheduling problem. We now show how to solve such a deadline scheduling problem in the divisible load framework.

In *deadline scheduling*, each job J_j has not only a release date r_j but also a deadline $\bar{d}_j$. The problem is then to find a schedule such that each job J_j is executed within its executable time interval $[r_j, \bar{d}_j]$. We consider the set of all job release dates and deadlines: $\{r_1, \ldots, r_n, \bar{d}_1, \ldots, \bar{d}_n\}$. We define an *epochal time* as a time value at which one or more points in this set occur; there are between 2 (when all jobs are released at the same date and have the same deadline) and $2n$ (when all job release dates and deadlines are distinct) such values. When ordered in absolute time, adjacent epochal times define a set of *time intervals*. We denote each time interval I_t by $I_t = [\inf I_t, \sup I_t[$. Finally, we denote by $\alpha_{i,j}^{(t)}$ the fraction of job J_j processed by machine M_i during the time interval I_t. In this framework, System (4.1) lists the constraints that should hold true in any valid schedule:

1. *release date*: job J_j cannot be processed before it is released (Equation (4.1a));
2. *deadline*: job J_j cannot be processed after its deadline (Equation (4.1b));
3. *resource usage*: during a time interval, a machine cannot be used longer than the duration of this time interval (Equation (4.1c));
4. *job completion*: each job must be processed to completion (Equation (4.1d)).

$$\begin{cases} (4.1a) \quad \forall i, \forall j, \forall t, \quad r_j \geq \sup I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ (4.1b) \quad \forall i, \forall j, \forall t, \quad \bar{d}_j \leq \inf I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ (4.1c) \quad \forall t, \forall i, \quad \sum_j \alpha_{i,j}^{(t)}.p_{i,j} \leq \sup I_t - \inf I_t \\ (4.1d) \quad \forall j, \quad \sum_t \sum_i \alpha_{i,j}^{(t)} = 1 \end{cases} \tag{4.1}$$

Lemma 2. *System* (4.1) *has a solution if, and only if, there exists a solution to the deadline scheduling problem.*

System (4.1) can be solved in polynomial time by any linear solver system as all its variables are rational. Building a valid schedule from any solution of System (4.1) is straightforward as for any time interval I_t, and on any machine M_i, the job fractions $\alpha_{i,j}^{(t)}$ can be scheduled in any order.

One may think that by applying a binary search on possible values of the objective value $\mathcal{F}$, one would be able to find the optimal maximum weighted flow, and an optimal schedule. However, a binary search on rational values will not terminate. By setting a limit on the precision of the binary search, the number of process iterations is bounded, and the quality of the approximation can be guaranteed. However, as we now show, we can adapt our search to always find the optimal in polynomial time.

Solving on a range of objective values

So far we have used System (4.1) to check whether our problem has a solution whose maximum weighted flow is no greater than some objective value $\mathcal{F}$. We now show that we can use it to check whether our problem has a solution for some particular *range* of objective values. Later we show how to divide the whole search space into a polynomial number of search ranges.

First, let us suppose that there exist two values $\mathcal{F}_1$ and $\mathcal{F}_2$, $\mathcal{F}_1 < \mathcal{F}_2$, such that the relative order of the release dates and deadlines, $r_1, \ldots, r_n, \bar{d}_1(\mathcal{F}), \ldots, \bar{d}_n(\mathcal{F})$, when ordered in absolute time, is independent of the value of $\mathcal{F} \in]\mathcal{F}_1; \mathcal{F}_2[$. Then, on the objective interval $]\mathcal{F}_1, \mathcal{F}_2[$, as before, we define an epochal time as a time value at which one or more points in the set $\{r_1, \ldots, r_n, \bar{d}_1(\mathcal{F}), \ldots, \bar{d}_n(\mathcal{F})\}$ occurs. Note that an epochal time which corresponds to a deadline is no longer a constant but an affine function in $\mathcal{F}$. As previously, when ordered in absolute time, adjacent epochal times define a set of *time intervals*, that we denote by $I_1, \ldots, I_{n_{\text{int}}(\mathcal{F})}$. The durations of time intervals are now affine functions in $\mathcal{F}$. Using these new definitions and notations, we can solve our problem on the objective interval $[\mathcal{F}_1, \mathcal{F}_2]$ using System (4.1) with the additional constraint that $\mathcal{F}$ belongs to $[\mathcal{F}_1, \mathcal{F}_2]$ ($\mathcal{F}_1 \leq \mathcal{F} \leq \mathcal{F}_2$), and with the minimization of $\mathcal{F}$ as the objective. This gives us System (4.2).

$$\text{MINIMIZE} \quad \mathcal{F} \quad \text{UNDER THE CONSTRAINTS}$$
$$\begin{cases} (4.2a) \quad \mathcal{F}_1 \leq \mathcal{F} \leq \mathcal{F}_2 \\ (4.2b) \quad \forall i, \forall j, \forall t, \quad r_j \geq \sup I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ (4.2c) \quad \forall i, \forall j, \forall t, \quad \bar{d}_j \leq \inf I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ (4.2d) \quad \forall t, \forall i, \quad \sum_j \alpha_{i,j}^{(t)}.p_{i,j} \leq \sup I_t - \inf I_t \\ (4.2e) \quad \forall j, \quad \sum_t \sum_i \alpha_{i,j}^{(t)} = 1 \end{cases} \tag{4.2}$$

Particular objectives

The relative ordering of the release dates and deadlines only changes for values of $\mathcal{F}$ where one deadline coincides with a release date or with another deadline. We call such a value of $\mathcal{F}$ a *milestone*[1]. In our problem, there are at most n distinct release dates and as many distinct deadlines. Thus, there are at most $\frac{n(n-1)}{2}$ milestones at which a deadline function coincides with a release date. There are also at most $\frac{n(n-1)}{2}$ milestones at which two deadline functions coincides (two affine functions intersect in at most one point). Let n_{q} be the number of distinct milestones. Then, $1 \leq n_{\mathrm{q}} \leq n^2 - n$. We denote by $\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_{n_{\mathrm{q}}}$ the milestones ordered by increasing values. To solve our problem we just need to perform a binary search on the set of milestones $\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_{n_{\mathrm{q}}}$, each time checking whether System (4.2) has a solution in the objective interval $[\mathcal{F}_i, \mathcal{F}_{i+1}]$ (except for $i = n_{\mathrm{q}}$ in which case we search for a solution in the range $[\mathcal{F}_{n_{\mathrm{q}}}, +\infty[$). There is a polynomial number of milestones and System (4.2) can be solved in polynomial time. Therefore:

Theorem 18. *$\langle R|r_j; div|\max w_j F_j\rangle$ is polynomial: minimizing the maximum weighted flow is a polynomial problem, in the divisible load model.*

4.7 Offline max-stretch optimization and Pareto optimality

We have recalled in Section 1.3, that a schedule is Pareto optimal if it is impossible to strictly decrease the "cost" of a job without strictly increasing that of another. Any non-Pareto-optimal schedule can thus be considered as non-efficient as a strictly a better usage of the resources is possible. In our setting, this means that the first maximum should be minimized, then the second should be minimized, and so on.

Sum-based metrics obviously do not suffer from the same flaw and always produce Pareto-optimal schedules. That is why one only need to consider the metrics $\max_j C_j$ *Pareto*, $F_{\max}$ *Pareto*, and $S_{\max}$ *Pareto*. These scheduling metrics are likely to be much more difficult (but also much more meaningful) than the previous ones as we must optimize not just the cost of the more constraining job but the cost of all the jobs.

Algorithm 5: Heuristic Pareto minimization of max-stretch on one machine.

```
FixedStretch ← ∅
FreeStretch ← {J_1, ..., J_n}
while FreeStretch ≠ ∅ do
    Compute the minimum maximum stretch S of the jobs in FreeStretch taking into account that
    for any job J_j such that (J_j, S_j) ∈ FixedStretch, J_j has exactly a stretch of S_j.
    foreach J_j ∈ FreeStretch do
        Let d̄_j ← r_j + S × p_j
    foreach (J_j, S_j) ∈ FixedStretch do
        Let d̄_j ← r_j + S_j × p_j
    Schedule Earliest Deadline First (EDF) all the jobs (breaking ties randomly); Let C_j be the
    completion time of job J_j under this schedule
    foreach J_j ∈ FreeStretch do
        if C_j = d̄_j then
            FreeStretch ← FreeStretch \ {J_j}
            FixedStretch ← FixedStretch ∪ {(J_j, S)}
foreach (J_j, S_j) ∈ FixedStretch do
    Let d̄_j ← r_j + S_j × p_j
Schedule Earliest Deadline First (EDF) all the jobs
```

[1] Labetoulle, Lawler, Lenstra, and Rinnooy Kan [90] called such a value a "critical trial value".

Heuristic Pareto minimization of max-stretch on one machine

Algorithm 5 is an obvious algorithm which recursively tries to minimize the stretch of jobs: first it minimizes the max-stretch, then the number of jobs whose stretch is equal to the max-stretch, then the maximum stretch of the other jobs, and so on. The next theorem states in which cases Algorithm 5 is known to produce Pareto optimal schedules for max-stretch minimization.

Theorem 19. *Algorithm 5 produces a Pareto optimal schedule for max-stretch minimization on one machine with preemption if at no iteration of the while loop there are two jobs whose deadlines, defined at Steps 6 and/or 8, are equal.*

We conjecture that Algorithm 5 always produces a Pareto optimal schedule for max-stretch minimization on one machine with preemption. This conjecture is based on the facts that 1) the function which associates to a schedule the vector of the stretch of the jobs, sorted in non-decreasing order, is a continuous function; 2) we believe that the set of the instances for which Theorem 19 holds is dense in the space of all instances.

Heuristic Pareto minimization of max weighted flow on unrelated machines

Here, we target the more general case of the max weighted flow as we will need to specifically look at the special case of max-flow minimization. Algorithm 6 presents the solution we propose for the general case. The solution for uni-processor case cannot be straightforwardly extended to the general case as the Earliest Deadline First algorithm is obviously not optimal for non-uniform machines. Once again we (try to) recursively optimize the max weighted flow of the jobs. We compute the best achievable max weighted flow for the jobs whose weighted flow is not yet fixed, and we (try to) minimize the number of jobs whose weighted flow is equal to this maximum. As always the objective max weighted flow gives a deadline per *FreeWeightedFlow* job, i.e., per job whose weighted flow has not yet been fixed. We first minimize the number of distinct deadlines d such that there always is a job whose deadline is d and which is completed at date d. Then we minimize the number of (problematic) jobs, i.e., of jobs which are completed at their deadline. We can show that Algorithm 6 is correct.

Lemma 3. *Algorithm 6 produces a valid schedule.*

Step 15 does not explicit how the set "S'_d" of jobs whose completion date equals the deadline should be computed, especially as we would like this set to be as small as possible. In fact, in the general case this problem is NP-complete, as claimed by the next theorem which states the complexity of the Pareto max-flow minimization, and thus of the general case.

Theorem 20. *The Pareto minimization of max-flow on unrelated machines, $\langle R|div|F_{\max} Pareto\rangle$, is NP-complete.*

As release dates do not appear in the above theorem, we actually proved that $\langle R|div|\max_j C_j Pareto\rangle$ is NP-complete. In fact we proved an even stronger result, that is that minimizing the number of jobs whose completion date is equal to the makespan is NP-complete on unrelated machines, and under the divisible model.

Minimum hitting set is equivalent to Minimum set cover [65]. Therefore, one of the best polynomial time algorithm to approximate Minimum hitting set is the greedy algorithm which at each step picks the element which belongs to the largest number of still un-hit subsets. This greedy algorithm has an approximation ratio of $1 + \ln|S|$ [85, 124], where $|S|$ is the size of the set.

We do not know what is the complexity of the Pareto minimization of the max-stretch. Seeing how efficient is the greedy heuristic for the minimum hitting set problem, we simply suggest to use it to solve in practice Step 15. Furthermore, one can easily see that when the set S_d at Step 14 is always reduced to a singleton, Algorithm 6 produces an optimal schedule. Therefore:

Theorem 21. *Algorithm 6 produces a Pareto optimal schedule for max weighted flow minimization on unrelated machines under the divisible load model if the set S_d at Step 14 is always reduced to a singleton.*

We believe that, in practice, the set S_d will always be reduced to a singleton, and thus that Algorithm 6 will always produce optimal schedules in practice. (Note that the case of jobs of same size and same release date is not a problem.)

Algorithm 6: Heuristic Pareto minimization of max weighted flow.

FixedWeightedFlow $\leftarrow \emptyset$
FreeWeightedFlow $\leftarrow \{J_1, ..., J_n\}$
while *FreeWeightedFlow* $\neq \emptyset$ **do**
    Compute the minimum max weighted flow $\mathcal{S}$ of the jobs in *FreeWeightedFlow* taking into account that for any job J_j such that $(J_j, \mathcal{S}_j) \in$ *FixedWeightedFlow*, J_j has exactly a stretch of $\mathcal{S}_j$
    foreach $J_j \in$ *FreeWeightedFlow* **do**
        $\bar{d}_j \leftarrow r_j + \mathcal{S} \times p_j$
    foreach $(J_j, \mathcal{S}_j) \in$ *FixedWeightedFlow* **do**
        $\bar{d}_j \leftarrow r_j + \mathcal{S}_j \times p_j$
    $\mathcal{D} \leftarrow \{\bar{d}_j \mid S_j \in$ *FreeWeightedFlow*$\}$
    foreach $d \in \mathcal{D}$ **do**
        In the set of time intervals defined by the release dates and deadlines (see Section 4.6), let I_{t_d} be the time interval ending at date d: $\sup I_{t_d} = d$
        Solve System (4.3) (which attempts to complete strictly before d all jobs of deadline d)

$$\text{MAXIMIZE} \quad \delta \quad \text{UNDER THE CONSTRAINTS}$$
$$\begin{cases} \forall i, \forall j, \forall t,\ r_j \geq \sup I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ \forall i, \forall j, \forall t,\ \bar{d}_j \leq \inf I_t \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ \forall t, \forall i,\ \sum_j \alpha_{i,j}^{(t)}.p_{i,j} \leq \sup I_t - \inf I_t \\ \forall j,\ \sum_t \sum_i \alpha_{i,j}^{(t)} = 1 \\ \forall i,\ \sum_{j|\bar{d}_j=d} \alpha_{i,j}^{(t)}.p_{i,j} \leq (\sup I_{t_d} - \inf I_{t_d}) - \delta \end{cases} \tag{4.3}$$

        if $\delta = 0$ **then**
            $S_d \leftarrow \{J_j \in$ *FreeWeightedFlow* $\mid \bar{d}_j = d\}$
            Compute a subset S'_d of S_d such that all jobs in S'_d have a max weighted flow of $\mathcal{S}$, and such that all the other jobs in S_d can simultaneously have a max weighted flow strictly smaller than $\mathcal{S}$.
            foreach $J_j \in S'_d$ **do**
                FreeWeightedFlow $\leftarrow$ *FreeWeightedFlow* $\setminus \{J_j\}$
                FixedWeightedFlow $\leftarrow$ *FixedWeightedFlow* $\cup \{(J_j, \mathcal{S})\}$
foreach $(J_j, \mathcal{S}_j) \in$ *FixedWeightedFlow* **do**
    $\bar{d}_j \leftarrow r_j + \mathcal{S}_j \times p_j$
Build a schedule according to the solution of Linear Program 4.1.

4.8 Online max-stretch optimization

In this section, we are no longer focusing on the theoretical offline version of our problem, but too its practical online version. In this section, we first improve a lower bound on the competitive ratio of online algorithms for max-stretch minimization established by Bender, Chakrabarti, and Muthukrishnan [15]. Then we present the two competitive algorithms that have previously been proposed in the literature [15, 16]. Last we highlight some practical limitations of these algorithms and propose new heuristics that circumvent these limitations.

Lower bound on the competitiveness of online algorithms

Theorem 22. *For three lengths of jobs, there is no* $\frac{1}{2}\Delta^{\sqrt{2}-1}$*-competitive preemptive online algorithm (on one machine) minimizing max-stretch.*

This result is an improvement from the bound of $\frac{1}{2}\Delta^{\frac{1}{3}}$ established by Bender, Chakrabarti, and Muthukrishnan [15]. In fact, we establish this new bound by doing a more precise analysis of the exact same adversary. In their proof, Bender, Chakrabarti, and Muthukrishnan implicitly assumed that the algorithm knew in advance the ratio Δ of the sizes of the largest and shortest jobs. As, in the next section, we recall that there exist some $O(\sqrt{\Delta})$-competitive algorithms, this means we have roughly bridged half of the gap between the previous lower bound and the best existing algorithms.

Competitive online heuristics

We have already seen in Section 4.3 that FCFS, the optimal algorithm for the online minimization of max-flow, is only Δ-competitive for the online minimization of max-stretch. This seemingly bad result is obviously partially explained by Theorem 22. We now recall two existing online algorithms for max-stretch minimization before introducing a new one. Bender, Muthukrishnan, and Rajaraman [16] defined, for any job J_j, a pseudo-stretch $\widehat{\mathcal{S}}_j(t)$:

$$\widehat{\mathcal{S}}_j(t) = \begin{cases} \frac{t-r_j}{\sqrt{\Delta}} & \text{if } 1 \leq p_j \leq \sqrt{\Delta}, \\ \frac{t-r_j}{\Delta} & \text{if } \sqrt{\Delta} < p_j \leq \Delta. \end{cases}$$

Then, they scheduled the jobs by decreasing pseudo-stretches, potentially preempting running jobs each time a new job arrives in the system. They demonstrated that this method is a $O(\sqrt{\Delta})$-competitive online algorithm.

Bender, Chakrabarti, and Muthukrishnan [15] had previously described another $O(\sqrt{\Delta})$-competitive online algorithm for max-stretch. This algorithm works as follows: each time a new job arrives, the currently running job is preempted. Then, they compute the optimal (offline) max-stretch $\mathcal{S}^*$ of all jobs having arrived up to the current time. Next, a deadline is computed for each job J_j: $\bar{d}_j(\mathcal{F}) = r_j + \alpha \times \mathcal{S}^*/p_j$ Finally, a schedule is realized by executing jobs according to their deadlines, using the *Earliest Deadline First* strategy. To optimize their competitive ratio, Bender *et al.* set their *expansion* factor α to $\sqrt{\Delta}$. For both heuristics, the ratio Δ of the sizes of the largest and shortest jobs submitted to the system is thus assumed to be known in advance.

When they designed their algorithm, Bender *et al.* did not know how to compute the (offline) optimal maximum stretch. This problem is now overcome. The main remaining problem with their solutions, from our point of view, is that such an algorithm tries only to optimize the stretch of the most constraining jobs. Indeed, such an algorithm may very easily schedule all jobs so that their stretch is equal to the objective, even if most of them could have been scheduled to achieve far lower stretches. This problem is far from being merely theoretical, as we will see in Section 4.10. As we recalled in the previous section, this problem is in fact common to all algorithms minimizing a *max* objective without attempting to produce Pareto optimal schedules. We would like to circumvent the above problem when designing new online heuristics.

Practical online heuristics

The basic online heuristic we could derive from our offline algorithm would be along the same line as the algorithm of Bender, Chakrabarti, and Muthukrishnan: each time a new job arrives we would preempt

the running jobs (if any), compute the optimal max-stretch, and schedule the jobs according to the solution of System 4.2. The solution of System 4.2 specifies what fraction of each job should be executed on each processor during each time interval. We would implement this solution by arbitrarily breaking the ties that may appear in each time interval.

Our first modification to this scheme is that, rather than computing the "optimal max-stretch", we compute the "best achievable max-stretch considering the decisions already made". In other words, we take into account our knowledge of which jobs were already (partially) executed, and when. The underlying idea being that we cannot change the past. Also, such an optimization will greatly simplify the linear system. This modification is implemented by making trivial modifications to System 4.2.

Our second modification to the above scheme is more important: we want to optimize more than the max-stretch. The first possibility is to use in an online framework our offline heuristic for the Pareto minimization of max-stretch. To do so, instead of using a binary search and System 4.2 to compute the best achievable max-stretch, we use Algorithm 6 where, at Step 4, we compute the best achievable max-stretch rather than the optimal one. This way we define our Online-Pareto heuristics.

Another possible approach is to specify that each job should be scheduled in a manner that minimizes its own stretch value, while maintaining the overall maximal stretch value obtained. For example, one could theoretically try to minimize the sum-stretch under the condition that the max-stretch be optimal. However, as we have seen, minimizing the sum-stretch is an open problem. So we consider a heuristic approach expressed by System (4.4). This system ensures that each job is completed no later than the deadline defined by the given max-stretch $\mathcal{S}^*$. Then, under this constraint, this system attempts to minimize an objective that resembles a rational relaxation of the sum-stretch (or more generally of the sum weighted flow) using, as an approximation of the completion time, the weighted sum of the average execution times of a job. As we do not know the precise time within an interval when a part of a job will be scheduled, we approximate it by the mean time of the interval. (This heuristic obviously offers no guarantee on the sum-stretch achieved.)

$$\text{Minimize} \quad \sum_{j=1}^{n} w_j \left(\left(\sum_t \left(\sum_{i=1}^{p} \alpha_{i,j}^{(t)} \right) \frac{\sup I_t(\mathcal{S}^*) + \inf I_t(\mathcal{S}^*)}{2} \right) - r_j \right) \quad \text{under the constraints}$$

$$\begin{cases} \text{(4.4a)} & \forall i, \forall j, \forall t, \quad r_j \geq \sup I_t(\mathcal{S}^*) \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ \text{(4.4b)} & \forall i, \forall j, \forall t, \quad \bar{d}_j(\mathcal{S}^*) \leq \inf I_t(\mathcal{S}^*) \Rightarrow \alpha_{i,j}^{(t)} = 0 \\ \text{(4.4c)} & \forall t, \forall i, \quad \sum_j \alpha_{i,j}^{(t)}.p_{i,j} \leq \sup I_t(\mathcal{S}^*) - \inf I_t(\mathcal{S}^*) \\ \text{(4.4d)} & \forall j, \quad \sum_t \sum_i \alpha_{i,j}^{(t)} = 1 \end{cases} \tag{4.4}$$

Using System 4.4 we obtain the following online algorithm. Each time a new job arrives:

1. Preempt the running job (if any).
2. Compute the best achievable max-stretch $\mathcal{S}^*$, considering the decisions already made.
3. With the deadlines and intervals defined by the max-stretch $\mathcal{S}^*$, solve System (4.4).

At this point, we define three variants to produce the schedule. The first, which we call Online, assigns work simply using the values found by the linear program for the α variables:

4. For a given processor P_i, and a given interval $I_t(\mathcal{S}^*)$, all jobs J_j that complete their fraction on that processor during that very interval (i.e., all jobs J_j such that $\sum_{t'>t} \alpha_{i,j}^{(t')} = 0$) are scheduled under the SWRPT policy in that interval. We call these jobs *terminal jobs* (for P_i and $I_t(\mathcal{S}^*)$). The non-terminal jobs scheduled on P_i during interval $I_t(\mathcal{S}^*)$ are only executed in $I_t(\mathcal{S}^*)$ after all terminal jobs have finished.

The second variant we consider, Online-EDF, attempts to make changes to the schedule at the processor level to improve the overall max- and sum-stretch attained:

4. Consider a processor P_i. The fractions $\alpha_{i,j}$ of the jobs that must be partially executed on P_i are processed on P_i under a list scheduling policy based on the following order: the jobs are ordered

according to the interval in which their share is completed (according to the solution of the linear program), with ties being broken by the SWRPT policy.

Finally, we propose a third variant, ONLINE-EGDF, that creates a global priority list:

4. The (active) jobs are processed under a list scheduling policy, using the strategy outlined in Section 4.2 to deal with restricted availabilities. Here, the jobs are totally ordered by the interval in which their *total work* is completed, with ties being broken by the SWRPT policy.

The validity of these heuristic approaches will be assessed through simulations in the section 4.10.

4.9 Summary of complexity results

Table 4.1 summarizes the main complexity results presented in this document as well as in related work. Minimizing $\max w_j F_j$ is polynomial as soon as divisibility or preemption is allowed whereas $\sum w_j F_j$ is always strongly NP-hard. $\sum F_j$ is easy only on simple settings (one processor with preemption of related processors with divisibility) and is strongly NP-hard in all other settings. The main problem whose complexity is still open is $\langle 1|r_j, pmtn|\sum S_j\rangle$ even if (as we already have mentioned in Section 4.5) Polynomial Time Approximation Scheme (PTAS) have been proposed for this problem.

	$model = \emptyset$	$model = pmtn$	$model = div$
$\langle 1\|r_j; model\|\max w_j F_j\rangle$	NP([15])	$\downarrow$	$\downarrow$
$\langle P\|r_j; model\|\max w_j F_j\rangle$	$\uparrow$	$\downarrow$	$\downarrow$
$\langle Q\|r_j; model\|\max w_j F_j\rangle$	$\uparrow$	$\downarrow$	$\downarrow$
$\langle R\|r_j; model\|\max w_j F_j\rangle$	$\uparrow$	P(Lin. Prog. [D1])	P(Lin. Prog. [D1])
$\langle 1\|r_j; model\|\sum F_j\rangle$	NP([94])	P(SRPT [4])	$\downarrow$
$\langle P\|r_j; model\|\sum F_j\rangle$	$\uparrow$	NP(Numerical-3DM [7])	$\downarrow$
$\langle Q\|r_j; model\|\sum F_j\rangle$	$\uparrow$	$\uparrow$	P(SRPT + Sec. 4.2)
$\langle R\|r_j; model\|\sum F_j\rangle$	$\uparrow$	$\uparrow$	NP(3DM, Sec. 4.4)
$\langle 1\|r_j; model\|\sum S_j\rangle$	NP(Sec. 4.5)	?	?
$\langle P\|r_j; model\|\sum S_j\rangle$	$\uparrow$	?	?
$\langle Q\|r_j; model\|\sum S_j\rangle$	$\uparrow$	?	?
$\langle R\|r_j; model\|\sum S_j\rangle$	$\uparrow$	?	NP(3DM, Sec. 4.5)
$\langle 1\|r_j; model\|\sum w_j F_j\rangle$	NP([94])	NP(Numerical-3DM [90])	$\nwarrow$
$\langle P\|r_j; model\|\sum w_j F_j\rangle$	$\uparrow$	$\uparrow$	$\uparrow$
$\langle Q\|r_j; model\|\sum w_j F_j\rangle$	$\uparrow$	$\uparrow$	$\uparrow$
$\langle R\|r_j; model\|\sum w_j F_j\rangle$	$\uparrow$	$\uparrow$	$\uparrow$

Table 4.1: Summary of complexity results.

4.10 Simulations

To evaluate the efficacy of various scheduling strategies when optimizing stretch-based metrics, we implemented a simulator using the SimGrid toolkit [92], based on the biological sequence comparison scenario. The application and platform models used in the resulting simulator are derived from our initial observations of the GriPPS system, described in Section 4.2. Our primary goal is to evaluate the proposed heuristics in realistic conditions that include partial replication of target sequence databases across the available computing resources. The remainder of this section outlines the experimental variables we considered and presents results describing the behavior of the heuristics in question under various parameterizations of the platform and application models.

Simulation settings

The platform and application models that we address in this work are quite flexible, resulting in innumerable variations in the range of potentially interesting combinations. To facilitate our studies,

we concretely define certain features of the system that we believe to be useful in describing realistic execution scenarios. We consider in particular six such features.

Platform size: Typically, a given biological database such as those considered in this work, would be replicated at various sites, at which comparisons against this database may be performed. Generally, the number of sites in a simulated system provides a basic measure of the aggregate power of the platform. This parameter specifies the exact number of sites in the simulated platform. Without loss of generality, we arbitrarily define each site to contain 10 processors.

Processor power: Our model assumes that all the processors at any given site are equivalent, and each processor is assumed to have access to all databases located there. Thus for each site, a single processor value represents the processing power at that site. We choose processor power values using the benchmark results we presented in [C12].

Number of databases: Applications such as GriPPS can accommodate multiple reference databases. Our model allows for any number of distinct databases to exist throughout the system.

Database size: We demonstrated in [C12] that the processing time needed to service a user request targeting a particular database varies linearly according to the number of sequences found in the database in question. We choose such values from a continuous range of realistic database sizes, with the job size for jobs targeting a particular database scaled accordingly.

Database availability: A particular database may be replicated at multiple sites, and a single site may host copies of multiple databases. We account for these two eventualities by associating with each database a probability of existence at each site. The same database availability applies to all databases in the system. We further ensure that each database is available on at least one site, and each site hosts at least one database.

Workload density: For a particular database, we define the workload density of a system to be the ratio, on average, of the aggregate job size of user requests against that database to the aggregate computational power available to serve such requests. Workload density expresses a notion of the "load" of the system. This parameter, along with the size of the database, define the frequency of job arrivals in the system.

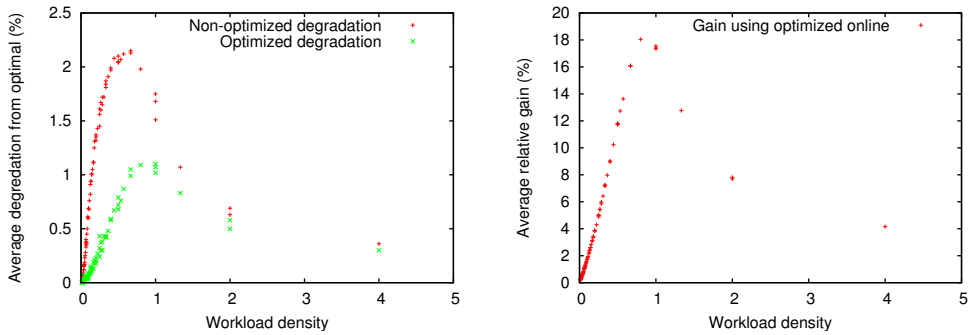
We define a *simulation configuration* as a set of specific values for each of these six properties. Once defined, concrete *simulation instances* are constructed by realizing random series for any random variables in the system. In particular, two models are created for each instance: a platform model and a workload model. The former is specified first by defining the appropriate number of 10-node sites and assigning corresponding processor power values. Next, a size is assigned to each database, and it is replicated according to the simulation's database availability parameter. Finally, the workload model is realized by first generating a series of jobs for each database, using a Poisson process for job inter-arrival times, with a mean that is computed to attain the desired workload density. The database-specific workloads are then merged and sorted to obtain the final workload. Jobs may arrive between the time at which the simulation starts and 15 minutes thereafter.

In this simulation study, we use empirical values observed in the GriPPS system logs to define a realistic range of database sizes and to generate appropriate values for processor speeds. The remaining four parameters – platform size, number of distinct databases, database availability, and workload density – are the simulation parameters that vary in our study. Later we will further discuss the specifics of the experimental design and our simulation results.

Optimization of the online heuristic

To motivate the variants of our online heuristic described in Section 4.8, we conduct a series of experiments to evaluate their effect. In particular, we consider a non-optimized version of the online heuristic, which stops after Step 2. We consider job workloads of average density varying between 0.0125 to 4.00, over a range of average job lengths between 15 and 60 seconds. For each job size/workload density combination evaluated, we simulate the execution of 5000 instances, recording the maximum and sum stretch of jobs in the workload achieved with both the optimized and non-optimized versions of the online heuristic. The max-stretch of each is then divided by the max-stretch achieved by the optimal

algorithm, yielding a degradation factor for both heuristics on that run. Since the optimal sum-stretch is not known, we observe the sum-stretch of the optimized online heuristic relative to the non-optimized version. Figures 4.3(a) and 4.3(b) present the max-stretch and sum-stretch results, respectively. In the first plot, the average max-stretch degradation, compared to the optimal result, for both versions of the heuristic over the 5000 runs of a given configuration is plotted against the workload density of that configuration. The second plot depicts the gain for the sum-stretch metric for the optimized heuristic, relative to the non-optimized version. These results strongly motivate the use of the optimizations encoded by the linear program depicted in System (4.4).



(a) Max-stretch degradation from optimal of both versions of the online heuristic

(b) Sum-stretch gain of the optimized version, relative to the non-optimized version

Figure 4.3: Comparison of the optimized and non-optimized versions of the online heuristic.

Simulation results and analysis

We have implemented in our simulator a number of scheduling heuristics that we plan to compare. First, we have implemented Offline, corresponding to the algorithm described in Section 4.6 that solves the optimal offline max-stretch problem. We have also implemented its Pareto version, Algorithm 6, under the name OfflinePareto. Three versions of the online heuristic are also implemented, designated as Online, Online-EDF, and Online-EGDF. Next, we consider the SWRPT, SRPT, and SPT heuristics discussed in Section 4.5. We did not implemented Online-Pareto because OfflinePareto is very computationally expensive. Then, we consider the two online heuristics proposed by Bender *et al.* that were briefly described in Section 4.8. We also include two greedy strategies. First, MCT ("minimum completion time") simply schedules each job as it arrives on the processor that would offer the best job completion time. The FCFS-Div heuristic extends this approach to take advantage of the fact that jobs are divisible, by employing all resources that are able to execute the job (using the strategy laid out in Section 4.2). Note that neither MCT nor FCFS-Div makes any changes to work that has already been scheduled. Finally, as a basic reference, we consider a list-scheduling heuristic with random job order denoted Rand. This heuristic works as follows: initially, we randomly build an order on the jobs that may arrive; then Rand list-schedules the jobs while using this list to define priorities, and while using the divisibility property. All the uni-processor heuristics (SWRPT, SRPT, SPT, and Bender *et al.*'s) are extended to the multi-processor case using the strategy previously described in Section 4.2.

As mentioned earlier, two of the six parameters of our model reflect empirical values determined by the study of the GriPPS system we reported in [C12]. Processor speeds are chosen randomly from one of the six reference platforms we studied, and we let database sizes vary continuously over a range of 10 megabytes to 1 gigabyte, corresponding roughly to GriPPS database sizes. Thus, our experimental results examine the behaviors of the aforementioned heuristics as we vary our four experimental parameters:
platforms of 3, 10, and 20 clusters (sites) with 10 processors each;
applications with 3, 10, and 20 distinct databases;
database availabilities of 30%, 60%, and 90% for each database; and
workload density factors of 0.75, 1.0, 1.25, 1.5, 2.0, and 3.0.[2]

[2]According to Frachtenberg and Feitelson [63] we fell into a classical pitfall of job scheduling evaluation. Following them, we should only have studied workloads whose density was smaller than 1. As we limited the running time of our

	Max-stretch			Sum-stretch		
	Mean	SD	Max	Mean	SD	Max
Offline	1.0000	0.0000	1.0000	1.4051	0.2784	2.6685
OfflinePareto	1.0000	0.0000	1.0000	1.2986	0.2605	3.5090
Online	1.0039	0.0145	1.2420	1.0458	0.0439	1.5069
Online-EDF	1.0040	0.0156	1.6886	1.0450	0.0432	1.5016
Online-EGDF	1.0331	0.0622	1.6613	1.0024	0.0052	1.1095
SWRPT	1.0386	0.0729	2.0566	1.0003	0.0014	1.0384
SRPT	1.0596	0.1027	2.1012	1.0048	0.0074	1.1179
SPT	1.0576	0.1032	2.1297	1.0020	0.0048	1.1263
Bender98	1.0415	0.0971	2.1521	1.0028	0.0075	1.1393
Bender02	2.9859	2.7071	23.5446	1.2049	0.3087	6.6820
FCFS-Div	5.1353	6.6792	65.9073	1.3767	0.7224	15.4213
MCT	38.4276	24.2626	156.3778	51.9606	36.5202	154.1519
Rand	4.6568	6.9107	87.9141	1.2355	0.4827	10.8549

Table 4.2: Aggregate statistics over all 162 platform/application configurations.

The resulting experimental framework has 162 configurations. For each configuration, 200 platforms and application instances are randomly generated and the simulation results for each of the studied heuristics is recorded. Table 4.2 presents the aggregate results from these simulations (finer-grained results based on various partitionings of the data may be found in our research report [D1]).

Above all, we note that the MCT heuristic – effectively the policy in the current GriPPS system – is unquestionably inappropriate for max-stretch optimization: MCT was over 38 times worse on average than the best heuristic. Its deficiency might arguably be tolerable on small platforms but, in fact, MCT yielded max-stretch performance over 16 times worse than the best heuristic in all simulation configurations. Even after addressing the primary limitation that the divisibility property is not utilized, the results are still disappointing: FCFS-Div is on average 5.1 times worse in terms of max-stretch than the best approach we found. One of the principal failings of the MCT and FCFS-Div heuristics is that they are non-preemptive. By forcing to wait a small task that arrives in a heavily loaded system, non-preemptive schedulers cause such a task to be inordinately stretched relative to large tasks that are already running.

Experimentally, we find that the first two of the three online heuristics we propose are consistently near-optimal (within 4‰ on average) for max-stretch optimization. The third heuristic, Online-EGDF, actually achieves consistently good sum-stretch (within 3‰ of the best observed sum-stretch), but at the expense of its performance for the max-stretch metric (within 4% of the optimal). This is not entirely surprising as this heuristic ignores a significant portion of the fine-tuned schedule generated by the linear program designed to optimize the max-stretch. Furthermore, our three online heuristics have far better sum-stretch than the OfflinePareto (which is on average almost 30% away of the best observed sum-stretch). This result validates our heuristic optimization of sum-stretch as expressed by Linear Program 4.4. OfflinePareto achieves a sum-stretch which is almost 8% smaller than that of Offline. This exemplifies what we previously stressed about the necessity to go beyond the simple optimization of the max-metric.

We also observe that SWRPT, SRPT, and SPT are all quite effective at sum-stretch optimization. Each is on average within 5‰ of the best observed sum-stretch for all configurations. In particular, SWRPT produces a sum-stretch that is on average 0.3‰ within the best observed sum-stretch, and attaining a sum-stretch within 4% of the best sum-stretch in all of the roughly 32,000 instances. However, it should be noted that these heuristics *may* lead to starvation. Jobs may be delayed for an arbitrarily long time, particularly when a long series of small jobs is submitted sequentially (the $(n+1)^{th}$ job being released right after the termination of the n^{th} job). Our analysis of the GriPPS application logs has revealed that such situations occur fairly often apparently due to automated processes that submit jobs at regular intervals. By optimizing max-stretch in lieu of sum-stretch, the possibility of starvation is

simulations and as we considered the stretch of all the jobs submitted, I believe that studying density factors greater than 1 correspond to studying load bursts. As our results are consistent whatever the density factor, I am confident our simulations are conclusive despite what may look as a methodological error.

[2] Bender98 results are limited to 3-cluster platforms, due to prohibitive overhead costs (discussed in Section 4.10).

	Max-stretch			Sum-stretch		
	Mean	SD	Max	Mean	SD	Max
OFFLINE	1.0000	0.0000	1.0000	1.0413	0.0593	1.6735
ONLINE	1.0016	0.0149	1.6344	1.0549	0.0893	1.8134
SWRPT	1.1316	0.2071	3.1643	1.0001	0.0009	1.0398
SRPT	1.1242	0.2003	3.0753	1.0139	0.0212	1.2576
SPT	1.1961	0.2667	3.9752	1.0229	0.0296	1.3573
BENDER98	1.1200	0.1766	2.5428	1.0194	0.0279	1.4466
BENDER02	3.5422	2.4870	21.4819	2.9872	1.9599	15.0019
FCFS-DIV	8.7762	9.1900	80.7465	6.8979	7.7409	88.2449
RAND	11.3059	11.1981	125.3726	5.8227	6.3942	68.0009

Table 4.3: Aggregate statistics for a single machine for all application configurations.

eliminated.

Next, we find that the BENDER98 and BENDER02 heuristics are not practically useful in our scheduling context. The results shown in Table 4.2 for the BENDER98 heuristic comprise only 3-cluster platforms; simulations on larger platforms were practically infeasible, due to the algorithm's prohibitive overhead costs. Effectively, for an n-job workload, the BENDER98 heuristic solves n optimal max-stretch problems, many of which are computationally equivalent to the full n-job optimal problem. In several cases the desired workload density required thousands of jobs, rendering the BENDER98 algorithm intractable. To roughly compare the overhead costs of the various heuristics, we ran a small series of simulations using only 3-cluster platforms. The results of these tests indicate that the scheduling time for a 15-minute workload was on average under 0.28 s for any of our online heuristics, and 0.54 s for the offline optimal algorithm (with 0.35 s spent in the resolution of the linear program and 0.19 s spent in the online phases of the scheduler); by contrast, the average time spent in the BENDER98 scheduler was 19.76 s. The scheduling overhead of BENDER02 is far less costly (on average 0.23 s of scheduling time in our overhead experiments), but in realistic scenarios for our application domain, the competitive ratios it guarantees are ineffective compared with our online heuristics for max-stretch optimization. Note that the bad performance of BENDER02 is not due to the way we adapt single-machine algorithms to unrelated machines configurations (see Section 4.2). Indeed, similar observations can be done when restricting to single-machine configurations (see Table 4.3).

Finally, we remark that the RAND heuristic is slightly better than the FCFS-DIV for both metrics. Moreover, RAND is only 24% away from the best observed sum-stretch on average. This leads us to think that the sum-stretch may not be a discriminating objective for our problem. Indeed, it looks as if, whatever the policy, any list-scheduling heuristic delivers good performance for this metric.

4.11 Conclusion

We have presented a polynomial-time algorithm to minimize the max-stretch in an offline settings, some heuristics to address the offline Pareto case, and some heuristics for the online framework. The theoretical part of our study enabled us to have an absolute estimate of the quality of our solutions. It enabled us to conclude from our simulations that not only our online heuristics were the most efficient, but also that they were close to optimal, which is not the case for the pre-existing guaranteed heuristics. To circumvent the drawbacks of max-stretch metrics we went beyond the simple max optimization by looking at Pareto optimization as well as the optimization of a second criteria (namely a relaxation of sum-stretch). The simulations proved that these approaches indeed significantly improve the quality of the schedules. Finally, our simulations also showed that the starvation prone sum-stretch metrics is not discriminating in our context has any list scheduling algorithm exhibits good performance for it. The sum-stretch metrics is thus questionable.

Chapter 5

References and personal publications

5.1 References

[1] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive task scheduling with parallelism feedback. In *Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York City, NY, USA, March 2006.

[2] A. Amoroso, K. Marzullo, and A. Ricciardi. Wide-area Nile: a case study of a wide-area data-parallel application. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 506–515. IEEE Computer Society Press, 1998.

[3] Eric Angel, Evripidis Bampis, and Fanny Pascual. Truthful algorithms for scheduling selfish tasks on parallel machines. *Theoretical Computer Science*, 369(1-3):157–168, 2006.

[4] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[5] K.R. Baker, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnooy Kan. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints. *Operations Research*, 31(2):381–386, March 1983.

[6] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15(4):319–330, 2004.

[7] Philippe Baptiste, Peter Brucker, Marek Chrobak, Christoph Durr, Svetlana A. Kravchenko, and Francis Sourd. The complexity of mean flow time scheduling problems with release times, 2006. Available at `http://arxiv.org/abs/cs/0605078`.

[8] Lawrence Barsanti and Angela C. Sodan. Adaptive job scheduling via predictive job resource allocation. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing (JSSPP 2006)*, volume 4376 of *Lecture Notes in Computer Science*, pages 115–140, 2007.

[9] Olivier Beaumont. *Nouvelles méthodes pour l'ordonnancement sur plates-formes hétérogènes.* Habilitation à diriger des recherches, Université de Bordeaux I, December 2004. `http://www.labri.fr/Perso/~obeaumon/habilitation.pdf`.

[10] 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'2006.* IEEE Computer Society Press, 2006.

[11] 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'2006.* IEEE Computer Society Press, 2006.

[12] 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, March 2005.

[13] Olivier Beaumont, Henri Casanova, Arnaud Legrand, Yves Robert, and Yang Yang. Scheduling divisible loads on star and tree networks: results and open problems. *IEEE Trans. Parallel Distributed Systems*, 16(3):207–218, 2005.

[14] Olivier Beaumont, Arnaud Legrand, and Yves Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Comput.*, 29(9):1121–1152, September 2003.

[15] 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.

[16] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Improved algorithms for stretch scheduling. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 762–771, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[17] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Approximation algorithms for average stretch scheduling. *J. of Scheduling*, 7(3):195–222, 2004.

[18] 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 Trans. Parallel Distributed Systems*, 14(4):369–382, 2003.

[19] Francine Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 12, pages 279–309. Morgan Kaufmann, 1999.

[20] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[21] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[22] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63(3):251–263, 2003.

[23] 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 *Supercomputing '96*. IEEE Computer Society Press, 1996.

[24] 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.

[25] Christophe Blanchet, Christophe Combet, Christophe Geourjon, and Gilbert Deléage. MPSA: Integrated System for Multiple Protein Sequence Analysis with client/server capabilities. *Bioinformatics*, 16(3):286–287, 2000.

[26] J. Blazewicz, J.K. Lenstra, and A.H. Kan. Scheduling subject to resource constraints. *Discrete Applied Mathematics*, 5:11–23, 1983.

[27] Jacek Błażewicz and Maciej Drozdowski. Distributed processing of divisible jobs with communication startup costs. *Discrete Appl. Math.*, 76(1-3):21–41, June 1997.

[28] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[29] R. C. Braun, Kevin T. Pedretti, Thomas L. Casavant, Todd E. Scheetz, C. L. Birkett, and Chad A. Roberts. Parallelization of local BLAST service on workstation clusters. *Future Generation Computer Systems*, 17(6):745–754, 2001.

[30] Andrey G. Bronevich and Wolfgang Meyer. Load balancing algorithms based on gradient methods and their analysis through algebraic graph theory. *Journal of Parallel and Distributed Computing*, 2:209–220, 2008.

[31] Peter Brucker. *Scheduling algorithms*. Springer, fourth edition edition, 2004.

[32] 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 Press.

[33] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[34] T. Carroll and D. Grosu. Strategyproof mechanisms for scheduling divisible loads in bus-networked distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 2008. To appear.

[35] Thomas E. Carroll and Daniel Grosu. Selfish multi-user task scheduling. *Fifth International Symposium on Parallel and Distributed Computing (ISPDC '06)*, pages 99–106, 2006.

[36] 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.

[37] Henri Casanova. Modeling large-scale platforms for the analysis and the simulation of scheduling strategies. In *Sixth Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*. IEEE Computer Society Press, 2004. Workshop held in conjunction with IPDPS.

[38] Henri Casanova. Network modeling issues for grid application scheduling. *International Journal of Foundations of Computer Science*, 6(2):145–162, 2005.

[39] Henri Casanova, Myungho Kim, James S. Plank, and Jack J. Dongarra. Adaptive scheduling for task farming with grid middleware. *Int. J. High Perform. Comput. Appl.*, 13(3):231–240, 1999.

[40] Henri Casanova, Arnaud Legrand, Dmitri Zagorodnov, and Francine Berman. Using simulation to evaluate scheduling heuristics for a class of applications in Grid environments. Research Report RR-1999-46, LIP, ENS Lyon, France, September 1999.

[41] Henri Casanova, Arnaud Legrand, Dmitri Zagorodnov, and Francine Berman. Heuristics for scheduling parameter sweep applications in Grid environments. In *Ninth Heterogeneous Computing Workshop*, pages 349–363. IEEE Computer Society Press, 2000.

[42] Chandra Chekuri and Sanjeev Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 297–305. ACM Press, 2002.

[43] 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. *J. Supercomput.*, 40(2):159–184, 2007.

[44] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[45] 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. doi:10.1016/S0167-8191(00)00087-9.

[46] CNRS. Le cnrs donne à la recherche française une puissance inégalée de calcul. `http://www2.cnrs.fr/presse/communique/1259.htm`.

[47] Gennaro Cordasco, Grzegorz Malewicz, and Arnold L. Rosenberg. Applying ic-scheduling theory to familiar classes of computations. In *Proceedings of PCGRID 2007*. IEEE, 2007.

[48] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[49] Albert Corominas, Wieslaw Kubiak, and Natalia Moreno Palli. Response time variability. *Journal of Scheduling*, 10(2):97–110, February 2007.

[50] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *Proceedings of ClusterWorld 2003*, 2003.

[51] Rodrigo Fernandes de Mello and Luciano José Senger. Model for simulation of heterogeneous high-performance computing environments. In Michel Daydé, José M.L.M. Palma, Álvaro L.G.A. Coutinho, Esther Pacitti, and João Correia Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2006*, volume 4395 of *Lecture Notes in Computer Science*, pages 107–119, 2007.

[52] Jack J. Dongarra, Emmanuel Jeannot, Erik Saule, and Zhiao Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 280–288, New York, NY, USA, 2007. ACM.

[53] Nikolaos D. Doulamis, Anastasios D. Doulamis, Emmanouel A. Varvarigos, and Theodora A. Varvarigou. Fair scheduling algorithms in grids. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1630–1648, 2007.

[54] 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).

[55] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks Approximation Algorithms. Chapman & Hall/CRC, 2004.

[56] Jeff Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[57] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task Scheduling in Multiprocessing Systems. *Computer*, 28(12):27–37, 1995.

[58] Paul Feautrier. Parametric integer programming. *RAIRO Rech. Opér.*, 22(3):243–268, September 1988.

[59] Dror G. Feitelson. Workload characterization and modeling (version 0.13), November 2007. `http://www.cs.huji.ac.il/~feit/wlmod/`.

[60] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 1–18, 1995.

[61] 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.

[62] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. J. Supercomput. Appl. High Perform. Comput.*, 11(2):115–128, 1997.

[63] Eitan Frachtenberg and Dror G. Feitelson. Pitfalls in parallel job scheduling evaluation. In D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *Lecture Notes in Computer Science*, pages 257–282, 2005.

[64] Noriyuki Fujimoto and Kenichi Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *International Conference on Parallel Processing*, pages 391–398, October 2003.

[65] M. Protasi G. Ausiello, A. D'Atri. Structure preserving reductions among convex optimization problems. *Journal of Computer and System Sciences*, 21(1):136–153, August 1980.

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

[67] T. Glatard, J. Montagnat, and X. Pennec. Probabilistic and dynamic optimization of job partitioning on a grid infrastructure. page 8 pp., 2006.

[68] GriPPS webpage at `http://gripps.ibcp.fr/`, 2005.

[69] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Seismic ray-tracing and Earth mesh modeling on various parallel architectures. *J. Supercomput.*, 29(1):27–44, July 2004.

[70] Tan Guang and Jarvis Stephen. A payment-based incentive and service differentiation scheme for peer-to-peer streaming broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 2008. To appear.

[71] Wong Han Min, Bharadwaj Veeravalli, and Gerassimos Barlas. Design and performance evaluation of load distribution strategies for multiple divisible loads on heterogeneous linear daisy chain networks. *Journal of Parallel and Distributed Computing*, 65(12):1558–1577, December 2005.

[72] Steve Hand. Virtualizing the data center with Xen. In *Keynote speach at EuroPar 2007*, 2007. `http://europar2007.irisa.fr/Files/Xen-EuroPar07.pps`.

[73] Marc Daniel Haunschild, Bernd Freisleben, Ralf Takors, and Wolfgang Wiechert. Investigating the dynamic behavior of biochemical networks using model families. *Bioinformatics*, 21(8):1617–1625, 2005.

[74] Ligang He, Stephen A. Jarvis, Daniel P. Spooner, David Bacigalupo, Guang Tan, and Graham R. Nudd. Mapping DAG-based applications to multiclusters with background workload. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, volume 2, pages 855–862, May 2005.

[75] René Henrion. Introduction to chance-constrained programming. `http://stoprog.org/index.html?SPIntro/intro2ccp.html`, 2004.

[76] Birgit Heydenreich, Rudolf Müller, and Marc Uetz. Games and mechanism design in machine scheduling: An introduction. *Production and Operations Management*, 16(4):437–454, 2007. Special Issue On E-Auctions for Procurement Operations.

[77] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In *Grid Computing (GRID 2000)*, volume 1971 of *Lecture Notes in Computer Science*, pages 214–227, 2000.

[78] 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'2004*. IEEE Computer Society Press, 2004.

[79] E. Huedo, R.S. Montero, and I.M. Llorente. Experiences on adaptive grid scheduling of parameter sweep applications. *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 28–33, 11-13 Feb. 2004.

[80] IDRIS. La nouvelle plate-forme de calcul intensif du cnrs. `http://www.idris.fr/IBM2008/Annonce-CNRS-IBM-070108.html`, 2008.

[81] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205, 2005.

[82] Dror Ironya, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Computing*, 64(9):1017–1026, 2004.

[83] Stephen A. Jarvis, Ligang He, Daniel P. Spooner, and Graham R. Nudd. The impact of predictive inaccuracies on execution scheduling. *Performance Evaluation*, 60(1-4):127–139, May 2005.

[84] Stavros A. Zenios John M. Mulvey, Robert J. Vanderbei. Robust optimization of large-scale systems. *Operations Research*, 43(2):264–281, 1995.

[85] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[86] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, May 2003.

[87] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM 2002 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 89–102. ACM Press, 2002.

[88] Hyoung Joong Kim, Gyu-In Jee, and Jang Gyu Lee. Optimal load distribution for tree network processors. *IEEE Trans. Aerosp. Electron. Syst.*, 32(2):607–612, April 1996.

[89] 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.

[90] Jacques Labetoulle, Eugene L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[91] Soojin Lee, Min-Kyu Cho, Jin-Won Jung, and Jai-Hoon Kim nd Weontae Lee. Exploring protein fold space by secondary structure prediction using data distribution method on grid platform. *Bioinformatics*, 20(18):3500–3507, 2004.

[92] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the 3rd IEEE Symposium on Cluster Computing and the Grid*, 2003.

[93] 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.

[94] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[95] Hui Li, Michael Muskulus, and Lex Wolters. Modeling job arrivals in a data-intensive grid. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing (JSSPP 2006)*, volume 4376 of *Lecture Notes in Computer Science*, pages 210–231, 2007.

[96] Y. Li and M. Mascagni. Grid-based Monte Carlo application. In *Proceedings of Grid Computing: GRID 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2002.

[97] 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 *Eight Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

[98] Grzegorz Malewicz, Arnold L. Rosenberg, and Matthew Yurkewych. Toward a theory for scheduling dags in internet-based computing. *IEEE Transactions on Computers*, 55(6):757–768, 2006.

[99] Nicole Megow. Performance analysis of on-line algorithms in machine scheduling. Diplomarbeit, Technische Universität Berlin, April 2002.

[100] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 37–48, New York, NY, USA, 2007. ACM.

[101] Perry L. Miller, Prakash M. Nadkarni, and N. M. Carriero. Parallel computation and FASTA: confronting the problem of parallel database search for a fast sequence comparison algorithm. *Computer Applications in the Biosciences*, 7(1):71–78, 1991.

[102] MPI Forum. MPI: A message passing interface standard, version 1.1. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[103] 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.

[104] M. G. Norman and P. Thanisch. Models of Machines and Computation for Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):103–117, 1993.

[105] The network simulator. `http://www.isi.edu/nsnam/ns/`.

[106] Tchimou N'Takpe, Frederic Suter, and Henri Casanova. A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms. *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on*, pages 35–35, 5-8 July 2007.

[107] G. R. Nudd and S. A. Jarvis. Performance-based middleware for grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):215–234, 2005.

[108] Andy Philpott. Stochastic programming introduction. `http://stoprog.org/index.html?spintroduction.html`.

[109] PIP/PipLib.
URL `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`.

[110] Joe Pitt-Francis, Alan Garny, and David Gavaghan. Enabling computer models of the heart for high-performance computers and the grid. *Philosophical Transactions of the Royal Society A*, 364(1843):1501–1516, June 2006.

[111] Kavitha Ranganathan and Ian Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1(1):53–62, 2003.

[112] Hélène Renard. *Équilibrage de charge et redistribution de données sur plates-formes hétérogènes*. PhD thesis, École normale supérieure de Lyon, December 2005.

[113] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Proceedings of the 19th ACM/SIAM Symposium on Discrete Algorithms (SODA 08)*, 2008.

[114] Krzysztof Rzadca, Denis Trystram, and Adam Wierzbicki. Fair game-theoretic resource management in dedicated grids. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 343–350, Washington, DC, USA, 2007. IEEE Computer Society.

[115] Jennifer M. Schopf and Francine Berman. Stochastic scheduling. *Supercomputing, ACM/IEEE 1999 Conference*, pages 48–48, 1999.

[116] Jennifer M. Schopf and Francine Berman. Using stochastic information to predict application behavior on contended resources. *Int. J. Found. Comput. Sci.*, 12(3):341–364, 2001.

[117] 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`.

[118] Uwe Schwiegelshohn and Ramin Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[119] Zhiao Shi, Emmanuel Jeannot, and Jack Dongarra. Robust task scheduling in non-deterministic heterogeneous computing systems. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster 2006)*. IEEE, 2006.

[120] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Science Press, 1995.

[121] Barbara Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, 12(2):294–299, May 1983.

[122] 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.

[123] Oliver Sinnen and Leonel A. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[124] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 435–441, New York, NY, USA, 1996. ACM Press.

[125] Wayne E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[126] J. R. Stiles, T. M. Bartol, E. E. Salpeter, and M. M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. In *Computational Neuroscience: Trends in Research, 1998*, pages 279–284. Plenum Press, 1998.

[127] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 2003.

[128] 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*, 0:195–202, 2006.

[129] A.G. Taylor and A.C. Hindmarsh. User documentation for KINSOL, a nonlinear solver for sequential and parallel computers. Technical Report UCRL-ID-131185, Lawrence Livermore National Laboratory, July 1998.

[130] 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.

[131] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[132] Sathish S. Vadhiyar and Jack J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[133] Nagavijayalakshmi Vydyanathan, Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, P. Sadayappan, and Joel Saltz. Scheduling of Tasks with Batch-shared I/O on Heterogeneous Systems. In *15th Heterogeneous Computing Workshop (HCW 2006)*. IEEE CS Press, April 2006.

[134] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'98)*. IEEE Computer Society Press, 1998.

[135] Rich Wolski, Daniel Nurmi, and John Brevik. An analysis of availability distributions in condor. In *NSF Next generation software (NGS) workshop (held in conjunction with IPDPS 2007)*. IEEE CS Press, 2007.

[136] L. Xiao, Y. Zhu, L. Ni, and Z. Xu. Incentive-based scheduling for market-like computational grids. *IEEE Transactions on Parallel and Distributed Systems*, 2008. To appera.

[137] Denise Sato Yamashita, Vinícius Amaral Armentano, and Manuel Laguna. Robust optimization models for project scheduling with resource availability cost. *J. of Scheduling*, 10(1):67–76, 2007.

[138] A. Zomaya and Y. Lee. A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 2008. To appear.

5.2 Publications

The publications are listed in reverse chronological order.

5.2.1 Books and book chapters

[A1] Yves Robert and Frédéric Vivien, editors. *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2008. To appear.

[A2] Yves Robert and Frédéric Vivien. Algorithmic issues in Grid computing. In *Algorithms and Theory of Computation Handbook*. Chapman and Hall/CRC Press, second edition, 2008. To appear.

[A3] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Divisible load scheduling. In *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2008. To appear.

[A4] Alain Darte, Yves Robert, and Frédéric Vivien. Loop parallelization algorithms. In *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques and Run Time Systems*, LNCS 1808, pages 141–171. Springer Verlag, 2001.

[A5] A. Darte, Y. Robert, and F.Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.

5.2.2 Articles in international refereed journals

[B1] 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.

[B2] Jean-François Pineau, Yves Robert, and Frédéric Vivien. The impact of heterogeneity on master-slave scheduling. *Parallel Computing*, 2008. To appear.

[B3] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Comments on "design and performance evaluation of load distribution strategies for multiple loads on heterogeneous linear daisy chain networks". *J. Parallel and Distributed Computing*, 2008. To appear.

[B4] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. *Journal of Scheduling*, 2008. To appear.

[B5] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling algorithms for data redistribution and load-balancing on master-slave platforms. *Parallel Processing Letters*, 17(1):61–77, 2007.

[B6] William Thies, Frédéric Vivien, and Saman Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):45 p., 2007.

[B7] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. *Journal of Systems Architecture*, 52(2):88–104, 2006.

[B8] Hélène Renard, Yves Robert, and Frédéric Vivien. Data redistribution algorithms for heterogeneous processor rings. *Int. Journal of High Performance Computing Applications*, 20(1):31–43, 2006.

[B9] Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. Automatic middleware deployment planning on clusters. *International Journal of High Performance Computing Applications*, 20(4):517–530, November 2006.

[B10] 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.

[B11] Frédéric Vivien and Nicolas Wicker. Minimal enclosing parallelepiped in 3D. *Computational Geometry: Theory and Applications*, 29(3):177–190, 2004.

[B12] Stéphane Genaud, Arnaud Giersch, and Frédéric Vivien. Load-balancing scatter operations for grid computing. *Parallel Computing*, 30(8):923–946, 2004.

[B13] Frédéric Vivien. On the Optimality of Feautrier's Scheduling Algorithm. *Concurrency and Computation: Practice and Experience*, 15(11-12):1047–1068, September 2003. Special issue on Euro-Par 2002.

[B14] Alain Darte, Rob Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):159–172, January 2002.

[B15] Pierre Boulet, Jack Dongarra, Yves Robert, and Frédéric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25:547–568, 1999.

[B16] Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert, and Frédéric Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2):197–213, 1999.

[B17] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti- and output-dependences. *Int. J. Parallel Programming*, 26(3):285–312, 1998.

[B18] Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert, and Frédéric Vivien. Retiming DAGs. *IEEE Trans. on computer-aided design of integrated circuits and systems*, 17(12):1319–1325, 1998.

[B19] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3-4):421–444, May 1998. Special issue on "Languages and Compilers for Parallel Computers".

[B20] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1-2):251–266, 1997.

[B21] Alain Darte and Frédéric Vivien. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. *International Journal of Parallel Programming*, 25(6):447–496, December 1997.

[B22] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Journal of Parallel Algorithms and Applications*, 12(1-3):83–112, 1997. Special issue on "Optimizing Compilers for Parallel Languages".

[B23] Alain Darte and Frédéric Vivien. Parallelizing Nested Loops with Approximation of Distance Vectors: A Survey. *Parallel Processing Letters*, 7(2):133–144, June 1997.

[B24] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, December 1997.

[B25] Alain Darte and Frédéric Vivien. Revisiting the decomposition of Karp, Miller and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.

5.2.3 Articles in international refereed conferences

[C1] 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. To appear.

[C2] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Offline and online scheduling of concurrent bags-of-tasks on heterogeneous platforms. In *10th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM 2008)*. IEEE Computer Society Press, 2008. To appear.

[C3] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling and data redistribution strategies on star platforms. In *PDP'2007, 15th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 288–295. IEEE Computer Society Press, 2007.

[C4] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling communication requests traversing a switch: complexity and algorithms. In *PDP'2007, 15th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 39–46. IEEE Computer Society Press, 2007.

[C5] 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.

[C6] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling multiple divisible loads on a linear processor network. In *ICPADS'2007, the 13th International Conference on Parallel and Distributed Systems*, 2007.

[C7] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A first step towards automatically building network representations. In *Proceedings of Euro-Par 2007*, volume 4641 of *LNCS*, pages 160–169, 2007.

[C8] 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.

[C9] 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.

[C10] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of biological requests. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 103–112, Cambridge, Massachusetts, USA, 2006. ACM Press.

[C11] Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. How should you structure your hierarchical scheduler? In *HPDC-15. 15th IEEE International Symposium on High Performance Distributed Computing*, pages 339–340 (Poster), Paris, France, June 19-23 2006. IEEE.

[C12] Arnaud Legrand, Alan Su, and Frédéric Vivien. Off-line scheduling of divisible requests on an heterogeneous collection of databanks. In *Proceedings of the 14th Heterogeneous Computing Workshop*, page (10 pages), Denver, Colorado, USA, April4 2005. IEEE Computer Society Press.

[C13] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *PDP'2004, 12th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 364–371. IEEE Computer Society Press, 2004.

[C14] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files from distributed repositories. In *Euro-Par-2004: International Conference on Parallel Processing*, LNCS 3149, pages 148–159. Springer Verlag, 2004.

[C15] Hélène Renard, Yves Robert, and Frédéric Vivien. Data redistribution algorithms for homogeneous and heterogeneous processor rings. In *International Conference on High Performance Computing HiPC'2004*, LNCS 3296, pages 123–132. Springer Verlag, 2004.

[C16] Hélène Renard, Yves Robert, and Frédéric Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. In *Euro-Par-2003: International Conference on Parallel Processing*, LNCS 2790, pages 148–159. Springer Verlag, 2003.

[C17] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Mapping and load-balancing iterative computations on heterogeneous clusters. In *Euro-PVM-MPI-2003: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS 2840, pages 586–594. Springer Verlag, 2003.

[C18] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files on heterogeneous clusters. In *Euro-PVM-MPI-2003: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS 2840, pages 657–660. Springer Verlag, 2003.

[C19] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Load-balancing iterative computations on heterogeneous clusters with shared communication links. In *PPAM-2003: Fifth International Conference on Parallel Processing and Applied Mathematics*, LNCS 3019, pages 930–937. Springer Verlag, 2003.

[C20] Stéphane Genaud, Arnaud Giersch, and Frédéric Vivien. Load-balancing scatter operations for grid computing. In *Proceedings of the 12th Heterogeneous Computing Workshop (HCW'2003)*. IEEE Computer Society Press, April 2003.

[C21] Frédéric Vivien. On the optimality of Feautrier's scheduling algorithm. In *Proceedings of Euro-Par 2002*, volume 2400 of *LNCS*, pages 299–308. Springer-Verlag, 2002.

[C22] Frédéric Vivien and Martin Rinard. Incrementalized Pointer and Escape Analysis. In *Proceedings of the ACM SIGPLAN'01 conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, Snowbird, UT, USA, June 2001.

[C23] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A Unified Framework for Schedule and Storage Optimization. In *Proceedings of the ACM SIGPLAN'01 conference on Programming Language Design and Implementation (PLDI)*, pages 232–242, Snowbird, UT, USA, June 2001.

[C24] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. A Constructive Solution to the Juggling Problem in Processor Array Synthesis. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 2000.

[C25] Alain Darte, Claude Diderich, Marc Gengler, and Frédéric Vivien. Scheduling the Computations of a Loop Nest with Respect to a Given Mapping. In *Proceedings of Euro-Par 2000*, volume 1900 of *LNCS*, pages 405–414, Munich, Germany, September 2000.

[C26] Pierre-Yves Calland, Anne Mignotte, Olivier Peyran, Yves Robert, and Frédéric Vivien. Retiming dags. In *IEEE/ACM Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems TAU'97*, pages 123–128. University of Texas at Austin, 1997.

[C27] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti and output dependences. In J. Fortes, C. Mongenet, K. Parhi, and V. Taylor, editors, *Application Specific Systems, Architectures and Processors*, pages 353–364. IEEE Computer Society Press, 1996.

[C28] Alain Darte and Frédéric Vivien. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT'96)*, pages 281–291, Boston, MA, USA, October 1996. IEEE Computer Society Press.

[C29] Alain Darte and Frédéric Vivien. On the Optimality of Allen and Kennedy's Algorithm for Parallelism Extraction in Nested Loops. In *Proceedings of Euro-Par'96*, volume 1123 of *LNCS*, pages 379–388, Lyon, France, August 1996. Springer-Verlag.

[C30] Alain Darte and Frédéric Vivien. A classification of nested loops parallelization algorithms. In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224, Paris, France, October 1995. IEEE Computer Society Press.

[C31] Alain Darte and Frédéric Vivien. Revisiting the Decomposition of Karp, Miller and Winograd. In P. Cappello, C. Mongenet, G.-R. Perrin, P. Quinton, and Y. Robert, editors, *Application Specific Array Processors*, pages 13–25, Strasbourg, France, July 1995. IEEE Computer Society Press.

5.2.4 Research reports

[D1] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. Research Report RR2008-08, LIP, École Normale Supérieure de Lyon, February 2008. This is a revised version of the LIP research report RR2006-19. Also available as INRIA research report 6002 `http://hal.inria.fr/inria-00108524`.

[D2] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling multiple divisible loads on a linear processor network. Research Report RR-6235, INRIA, 2007.

[D3] 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 RR-6401, INRIA, 2007.

[D4] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A first step towards automatically building network representations. Research Report RR-6133, INRIA, February 2007. Also available as LIP research report 2007-08.

[D5] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Comments on "design and performance evaluation of load distribution strategies for multiple loads on heterogeneous linear daisy chain networks". Research Report RR-6123, INRIA, February 2007. Also available as LIP research report 2007-07.

[D6] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling and data redistribution strategies on star platforms. Research report RR-6005, INRIA, 2006. `http://hal.inria.fr/inria-00108518`. Also available as LIP research report RR2006-23.

[D7] 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.

[D8] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling communication requests traversing a switch: complexity and algorithms. Research report RR2006-25, LIP, ENS Lyon, 2006. Also available as LIP research report.

[D9] Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. Research report RR-6053, INRIA, 2006. `http://hal.inria.fr/inria-00117050`. Also available as LIP research report RR2006-39.

[D10] 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.

[D11] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of biological requests. Technical Report LIP 2005-48, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, October 2005. Also available as INRIA research report 5724.

[D12] Eddy Caron, Pushpinder Kaur Chouhan, Holly Dail, and Frédéric Vivien. Automatic middleware deployment planning on clusters. Research report 2005-50, Laboratoire de l'Informatique du Parallélisme (LIP), October 2005. Revised version of LIP Research Report 2005-26.

[D13] Arnaud Legrand, Alan Su, and Frédéric Vivien. Off-line scheduling of divisible requests on an heterogeneous collection of databanks. Research report 5386, INRIA, November 2004. Also available as LIP, ENS Lyon, research report 2004-51.

[D14] Hélène Renard, Yves Robert, and Frédéric Vivien. Data redistribution algorithms for homogeneous and heterogeneous processor rings. Research Report 5207, INRIA, May 2004. Also available as LIP, ENS Lyon, research report 2004-28.

[D15] Arnaud Giersch, Yves Robert, and Frédérc Vivien. Scheduling tasks sharing files from distributed repositories (revised version). Research Report 5124, INRIA, February 2004. Also available as LIP, ENS Lyon, research report 2004-04.

[D16] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Load-balancing iterative computations in heterogeneous clusters with shared communication links. Technical Report 2003-23, LIP, April 2003.

[D17] Arnaud Giersch, Yves Robert, and Frédérc Vivien. Scheduling tasks sharing files from distributed repositories. Research Report 4976, INRIA, October 2003. Also available as LIP, ENS Lyon, research report 2003-49.

[D18] Arnaud Giersch, Yves Robert, and Frédérc Vivien. Scheduling tasks sharing files on heterogeneous clusters. Research Report 4819, INRIA, May 2003. Also available as LIP, ENS Lyon, research report 2003-28.

[D19] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédérc Vivien. Load-balancing iterative computations on heterogeneous clusters with shared communication links. Research Report 4800, INRIA, April 2003. Also available as LIP, ENS Lyon, research report 2003-23.

[D20] Stéphane Genaud, Arnaud Giersch, and Frédéric Vivien. Load-balancing scatter operations for grid computing. Research Report 4770, INRIA, March 2003. Also available as LIP, ENS Lyon, reserach report 2003-17.

[D21] Hélène Renard, Yves Robert, and Frédérc Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. Research Report 4745, INRIA, February 2003. Also available as LIP, ENS Lyon, research report 2003-12.

[D22] Frédéric Vivien and Nicolas Wicker. Minimal Enclosing Parallelepiped in 3D. Research report RR-2002-49, Laboratoire de l'Informatique du Parallélisme (LIP), December 2002. Also INRIA Research Report RR-4685.

[D23] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A constructive solution to the juggling problem in systolic array synthesis. Research report RR1999-15, LIP, ENS-Lyon, 1999.

[D24] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. Research report 97–17, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, June 1997. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR97/RR97-17.ps.Z`.

[D25] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. Research report 96–34, Laboratoire de l'informatique de parallélisme (LIP), École normale supérieure de Lyon, France, November 1996. `ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR96/RR96-34.ps.Z`.

[D26] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Research report 96-06, LIP, ENS-Lyon, France, April 1996. Published in the International Journal of Parallel Programming 25(6):447–496, 1997.

[D27] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Research report 96-05, LIP, ENS-Lyon, France, February 1996. Extended version of Europar'96.

[D28] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Research report 96-34, LIP, ENS-Lyon, France, November 1996.

[D29] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Research report 94-24, LIP, ENS-Lyon, France, September 1994.

Appendix A

Curriculum vitæ

– À mille sesterces le menhir, ça nous fait donc deux mille sesterces !
– Toi y en a compter vite !
– Moi y en a être habitué; moi y en a appris beaucoup de choses dans grande école.
Une aventure d'Astérix le gaulois: Obélix et compagnie, René Goscinny et Albert Uderzo.

A.1 Formation et parcours professionnel

Formation

- Thèse de **Doctorat** de l'École normale supérieure de Lyon, soutenue le 17 décembre 1997, intitulée « Détection de parallélisme dans les boucles imbriquées », et encadrée par Alain Darte et Yves Robert. Rapporteurs: Paul Feautrier et Robert Schreiber. Membres du jury: Philippe Chrétienne, Alain Darte, Paul Feautrier, Catherine Mongenet, Sanjay Rajopadhye, et Yves Robert.
- **DEA** d'informatique de l'École normale supérieure de Lyon, juin 1994, effectué sous la direction d'Alain Darte et intitulé: « Ordonnancement multidimensionnel pour les équations récurrentes uniformes et affines ».
- **Agrégation de mathématiques**, juin 1995.
- Élève de l'École normale supérieure de Lyon, 1991-1995.

Parcours professionnel

Situation actuelle: chargé de recherche INRIA (CR1), en poste dans le laboratoire LIP de l'École normale supérieure de Lyon, France.

Établissements	Fonctions et statuts	Début	Fin
INRIA	Chargé de recherche	1er sep. 2002	-
MIT	*Visiting Scientist*	15 jan. 2000	15 déc. 2000
Université Louis Pasteur Strasbourg, France	Maître de conférences	1er sep 1998	31 août 2002

A.2 Encadrement d'activités de recherches

A.2.1 (Co-)encadrement de thèses

- Matthieu Gallet a commencé sa thèse en septembre 2006. J'assure l'intégralité de son encadrement. Matthieu s'intéresse à l'utilisation de la réplication des calculs pour l'accélération de l'exécution des graphes de tâches.
- Jean-François Pineau a commencé sa thèse en septembre 2005. J'assure 50% de son encadrement (Yves Robert étant l'autre encadrant). Jean-François s'intéresse à l'impact des communications sur l'algorithmique pour plates-formes hétérogènes et distribuées.
- Hélène Renard: thèse soutenue en décembre 2005 et intitulée « Équilibrage de charge et redistribution de données sur plates-formes hétérogènes » (étude de l'ordonnancement des algorithmes itératifs sur plates-formes hétérogènes et distribuées). J'ai assuré 50% de l'encadrement de cette thèse (Yves Robert étant l'autre encadrant). Hélène est actuellement maître de conférences à l'École Polytechnique de l'Université de Nice - Sophia Antipolis.
- Arnaud Giersch: thèse soutenue en décembre 2004 et intitulée « Ordonnancement sur plates-formes hétérogènes de tâches partageant des données ». J'ai assuré 50% de l'encadrement de la deuxième moitié de cette thèse (Stéphane Genaud étant l'autre encadrant). Arnaud est actuellement maître de conférences à l'IUT de Belfort-Montbéliard.
- Nicolas Wicker: thèse soutenue en décembre 2002 et intitulée « Détermination du nombre de classes: application aux gènes et aux protéines ». J'ai assuré 50% de l'encadrement de la troisième et dernière année de thèse (Olivier Poch étant l'autre encadrant). Nicolas a effectué sa thèse en bioinformatique au sein d'une équipe de biologie structurale. Ensemble, nous avons travaillé sur l'approximation, par un catalogue de formes prédéfinies, de la « surface » d'une protéine. Nicolas est actuellement maître de conférences à l'Université Louis Pasteur, Strasbourg.

A.2.2 (Co-)encadrement de stages de DEA/M2 recherche

- Clément Rezvoy, soutenance prévue en juin 2007. Le stage de Clément a pour but la parallélisation de la construction de la base de données ProDom, base de données de domaines de protéines construite automatiquement à partir de bases de séquences. J'assure 50% de son encadrement, l'autre co-encadrant étant Daniel Khan, bioinformaticien du projet Hélix (Université Claude Bernard, Lyon).
- Veronika Rehn: « Scheduling and data redistribution strategies on star platforms », juin 2006. J'ai assuré 33% de son encadrement (Loris Marchal et Yves Robert étant les deux autres encadrants).
- Matthieu Gallet: « Ordonnancement de requêtes commutées », juin 2006. J'ai assuré 50% de l'encadrement de ce stage (Yves Robert étant l'autre encadrant).
- Jean-François Pineau: « Ordonnancement de tâches indépendantes sur plate-forme maître-esclave hétérogène: modèles hors-ligne et à la volée », juin 2005. J'ai assuré 50% de l'encadrement de ce stage (Yves Robert étant l'autre encadrant).
- Ridha Kouache: « Durées de vie et compression mémoire », juin 2002. J'ai assuré 50% de l'encadrement de ce stage (Philippe Clauss étant l'autre encadrant).

A.2.3 Encadrement de stage post-doctoral

- Lionel Eyraud effectue en 2006-2007 un stage post-doctoral d'un an dans l'équipe. Il s'intéresse à l'acquisition automatique d'une modélisation du réseau d'interconnexion reliant un ensemble de ressources de calcul.

A.3 Responsabilités collectives

A.3.1 Encadrement d'équipe

Je suis responsable du projet GRAAL depuis le 1er juillet 2006 (7 permanents, 1 post-doctorant, 7 doctorants et 6 ingénieurs au 1er janvier 2007).

J'ai été responsable de l'enseignement de l'informatique et du tutorat en DEUG MIAS à l'Université Louis Pasteur, Strasbourg, en 2001-2002.

A.3.2 Participation à des commissions

- Conseil de laboratoire du LIP depuis 2003 (membre élu jusqu'à mi-2006, puis membre de droit).
- Commission des emplois scientifiques de l'INRIA Rhône-Alpes pour les campagnes de recrutement de 2003 à 2006. Cette commission est responsable des recrutements en délégations, détachements, post-docs et professeurs invités.
- Commission de recrutement pour les bourses de thèse CORDI de l'INRIA Rhône-Alpes depuis 2006.
- Commission de spécialistes (27e section) de l'École normale supérieure de Lyon en 2005-2006 (suppléant).
- Commission de spécialistes (27e section) de l'Université Louis Pasteur, Strasbourg, en 2001-2002.
- Conseil de l'UFR de mathématique et d'informatique de l'Université Louis Pasteur, Strasbourg, en 2001-2002.

A.3.3 Examinateur au concours d'entrée aux ENS

J'ai été examinateur au concours d'entrée aux Écoles normales supérieures (Ulm, Lyon, Cachan) en 2003, 2004 et 2005 pour l'épreuve orale d'informatique fondamentale. J'ai conçu un tiers des sujets de ces épreuves (nous étions trois examinateurs).

A.3.4 Fonctions d'évaluation scientifique

Comité éditiorial

Depuis le 1er août 2005 je suis membre du comité éditorial du journal *Parallel Computing*.

Comités de programme de conférences

- EuroPDP 2008: *16th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Toulouse, France, février 2008.
- CoreGRID Workshop on Grid Middleware 2007, Dresden, RFA, 25-26 juin; ce *workshop* aura lieu conjointement avec la conférence ISC'07.
- Grid2007 : *8th IEEE International Conference on Grid Computing*, Austin, Texas, USA, 19-21 septembre 2007.
- *Workshop on Scheduling for Parallel Computing*, Gdansk, Pologne, septembre 2007; ce *workshop* aura lieu conjointement avec la conférence PPAM 2007.
- PMGC 2007 : *Workshop on Programming Models for Grid Computing*, Rio de Janeiro, Brésil, mai 2007; ce *workshop* aura lieu conjointement avec la conférence CCGrid 2007.
- IPDPS 2007: *21st IEEE International Parallel & Distributed Processing Symposium*, 26-30 mars 2007, Long Beach, Californie, USA.
- EuroPDP 2007: *15th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Naples, Italie, février 2007.
- HiPC 2006 : *13th International Conference on High Performance Computing*, Bangalore, Inde, décembre 2006.
- RenPar 2006 : 17e Rencontres francophones du Parallélisme, Perpignan, France, octobre 2006.
- Grid2006 : *7th IEEE International Conference on Grid Computing*, Barcelone, Espagne, septembre 2006.
- HCW 2006: *15th Heterogeneous Computing Workshop*, Rhodes, Grèce, avril 2006; ce *workshop* a eut lieu conjointement avec la conférence IPDPS.
- CoreGRID Workshop on Grid Middleware 2006, Dresde, Allemagne, 28-29 aoAût 2006; ce *workshop* a eut lieu conjointement avec la conférence Euro-Par.
- Grid 2005: *6th IEEE/ACM International Workshop on Grid Computing*; ce *workshop* a eu lieu conjointement avec la conférence SuperComputing'05 à Seattle, États-Unis, en novembre 2005.
- ISPA'05 : *International Symposium on Parallel and Distributed Processing and Applications*, Nanjing, Chine, novembre, 2005.
- *Workshop on scheduling for parallel computing*, Poznan, Pologne, septembre, 2005; ce *workshop* a eu lieu conjointement avec la conférence PPAM 2005.
- PPoPP 2003: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, États-Unis, juin 2003.

Évaluation de projets

En 2005, j'ai évalué un projet de recherche dans le cadre du *Inovational Research Incentives Scheme - Veni programme* de l'organisation néerlandaise pour la recherche scientifique (NWO).

A.3.5 Organisation de colloques

- Organisation de l'École de Printemps d'Informatique Théorique (EPIT 2007) avec Yves Robert, du 3 au 8 juin 2007, sur le thème de « Ordonnancement ».

- Organisation du colloque « Compilation et Parallélisation Automatique » avec Catherine Mongenet (ICPS-LSIIT, ULP) et Yves Robert (LIP, ENS Lyon), colloque qui a réuni du 18 au 20 octobre 1999 la communauté française de recherche en parallélisation automatique.

A.4 Participation à des projets et collaborations

A.4.1 Participation à des projets nationaux et internationaux

MIT-France (2007): responsable de la collaboration avec Saman Amarasinghe (MIT, USA) autour de l'ordonnancement de flots de calculs pour le processeur multicore Cell.

RTRA Innovations en infectiologie (2007-2011): responsable pour le projet GRAAL.

ANR Alpage (2006-2008): membre.

CoreGRID (2004-2008): Réseau d'excellence (NoE) européen. Responsable scientifique du partenaire « CNRS ». Responsable d'une sous-tâche de l'institut virtuel *Resource Management and Scheduling*.

Ragtime (2003-2006): projet de la région Rhône-Alpes sur la « Grille pour le Traitement d'Information Médicale ». Responsable du thème « Gestion des ressources de calculs ».

A.4.2 Relations académiques

France

- Projet Hélix, Université Claude Bernard, Lyon 1 (2006-): co-encadrement avec Daniel Khan du stage de Master 2 recherche de Clément Rezvoy.

- Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection, Illkirch (2002-2005): co-encadrement de la thèse d'Arnaud Giersch.

- Institut de Génétique et de Biologie Moléculaire et Cellulaire, Strasbourg (2001-2003): co-encadrement de la thèse de Nicolas Wicker.

- Projets Mescal et Moais du Laboratoire d'Informatique de Grenoble et projet Algorille du Loria, Nancy (2006-) autour du travail de Lionel Eyraud.

États-Unis

- Université du Tennessee à Knoxville (1997 et 2006-). Voir la section mobilité pour l'année 1997. Travaux en 2006, dans le cadre de la thèse de Jean-François Pineau, sur le produit de matrice sur plates-formes maîtres-esclaves hétérogènes.[]

- MIT (2001-). Voir les sections A.4.1 et A.4.3.

A.4.3 Mobilité

(Séjours de plus d'un mois hors du laboratoire de rattachement.)

Mi-janvier 2000 à mi-décembre 2000: séjour[1] dans le laboratoire d'informatique (LCS) du **Massachusetts Institute of Technology** (MIT) dans l'équipe de Martin Rinard au sein du *Compiler and Architecture Group*. Travaux avec Martin Rinard sur l'analyse de programmes Java [C22] : nous avons transformé une analyse sensible au flot de contrôle des référencements et échappements en une analyse incrémentale 1) analysant le minimum (du programme) nécessaire à l'obtention des informations requises pour décider de la validité d'une optimisation; 2) n'essayant d'optimiser que les "régions"dont l'optimisation permettrait un gain de performances significatif. Travaux avec Saman Amarasinghe et William Thies sur la réduction de la mémoire utilisée par les programmes au moyen de "vecteurs d'occupation affines"[C23].

Juin 1997: séjour d'un mois à l'**université du Tennessee à Knoxville**, USA, dans l'équipe de Jack Dongarra. Travaux théoriques et expérimentaux sur l'ordonnancement, sur un réseau hétérogène d'ordinateurs, des tuiles issues du partitionnement de boucles imbriquées.

Juillet-octobre 1996: séjour de trois mois dans les laboratoires de recherche de **Hewlett-Packard** à Palo Alto, Californie. Au sein du groupe *Compiler and Architecture Research* dirigé par Bob Rau, j'ai participé à la réalisation du logiciel *PICO* (*Program In, Chip Out*) qui est un générateur automatique de circuits dédiés. PICO prend en entrée un programme dont une grande partie du temps de calcul est due à l'exécution d'un nid de boucles. PICO génère automatiquement un « chip » en partie formé d'un réseau de microprocesseurs qui est construit sur mesure pour accélérer l'exécution du nid de boucles.

A.4.4 Participation à des développements logiciels

- *MIT-Flex* : implantation au sein de ce compilateur universitaire de Java d'une analyse incrémentale (voir aussi la section *Mobilité*) des référencements et échappements. Cette analyse représente 20,000 lignes de code (dans un compilateur de plus de 200,000 lignes) et environ six mois de travail. Le compilateur est distribué sous licence GPL : `http://flex-compiler.lcs.mit.edu`.
- *PICO* : implantation pendant trois mois en 1996 de transformations de programmes au sein du compilateur SUIF dans le cadre du projet *PICO*; le programme *PICO* était interne à Hewlett-Packard (voir aussi la section *Mobilité*) il est maintenant développé et commercialisé par la compagnie Synfora.

A.5 Activités d'enseignement

Le tableau A.1 présente l'ensemble des enseignements que j'ai dispensé depuis ma thèse, soit en tant que maître de conférences à Strasbourg, soit en tant que chargé de recherche à Lyon. Tous les volumes horaires sont exprimés en heures-équivalent-TD.

Acronymes: IUP = Institut Universitaire Professionalisé (IUP1 = L2, IUP2 = L3); EPG = École de Physique du Globe (EPG 3 = M2).

[1] Le séjour couvre le deuxième semestre de l'année universitaire 1999-2000 et le premier de 2000-2001. J'ai effectué l'intégralité de mon service d'enseignement au premier (resp. deuxième) semestre pour l'année universitaire 1999-2000 (resp. 2000-2001).

Année	Intitulé	Type	Public	Heures éq. TD
2006-2007	Ordonnancement sur plates-formes hétérogènes	Cours	M2 rech.	42
2005-2006	Ordonnancement sur plates-formes hétérogènes	Cours	M2 rech.	42
2004-2005	Algorithmique sur la grille	Cours	M2 rech.	12
2001-2002	Algorithmique avancée	Cours, TD et TP	IUP 2	61
	Architecture des ordinateurs	Cours, TD et TP	DEUG MIAS 2	66
	Pratique de la programmation	TP	IUP 1	67
	Techniques de parallélisation	Cours	DEA	12
2000-2001	Introduction à la programmation	TD et TP	DEUG MIAS 1	68
	Programmation fonctionnelle et logique	Cours, TD et TP	IUP 2	57
	Pratique de la programmation	TP	IUP 1	67
1999-2000	Téléinformatique et protocoles	TD et TP	IUP 2	60
	Programmation fonctionnelle et logique	Cours, TD et TP	IUP 2	63
	Algorithmique et programmation	TD et TP	IUP 1	68
	Initiation au calcul parallèle	Cours	EPG 3	18
	Techniques de parallélisation	Cours	DEA	6
1998-1999	Téléinformatique et protocoles	TD et TP	IUP 2	116
	Programmation fonctionnelle et logique	Cours, TD et TP	IUP 2	63
	Initiation au calcul parallèle	Cours	EPG 3	18
	Techniques de parallélisation	Cours	DEA	6

Table A.1: Enseignements dispensés depuis le recrutement.