N° d'ordre: 011

ECOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

Habilitation à diriger des recherches

presented by

Anne BENOIT

to obtain the HABILITATION À DIRIGER DES RECHERCHES
in *Computer Science* from Ecole Normale Supérieure de Lyon.

Scheduling Pipelined Applications: Models, Algorithms and Complexity

Defense date: July 8, 2009

Defense committee:

President	Mme Alix Munier	Professor, LIP6, Paris, France
Reviewers	M. Marco Danelutto	Professor, University of Pisa, Italy
	M. Charles Leiserson	Professor, MIT, USA
	M. Uwe Schwiegelshohn	Professor, University of Dortmund, Germany
Members	M. Yves Robert	Professor, ENS Lyon, France
	M. Denis Trystram	Professor, IMAG, Grenoble, France

Contents

Foreword

This document summarizes my research work since the end of my PhD thesis, in June 2003. It represents six years of work, two at the University of Edinburgh in Scotland, and four at ENS in Lyon.

First of all I would like to thank the members of my defense committee. In particular I thank Marco Danelutto, Charles Leiserson and Uwe Schwiegelshohn for their rewarding reports. I also thank Denis Trystram and Alix Munier for accepting to be part of the committee. Special thanks go to the last committee member, Yves Robert, with whom I have spent the last four years sharing these exciting research projects, and who supported me while writing this habilitation.

I would like to thank my colleagues at the University of Edinburgh, where I first discovered the "algorithmic skeletons" and scheduling challenges. It was a real pleasure to work with Murray Cole, Stephen Gilmore and Jane Hillston during my two years of post-doctoral research position in Edinburgh.

There are many more persons whom I want to thank, in particular all those with whom I shared some work or research discussions. All LIP members are extremely supportive and very friendly, and it is a real pleasure to work with them. Also, it was really rewarding to meet many different colleagues from all around the world, during various international workshops, conferences, and collaboration trips.

Finally, I do not forget all the friends with whom I shared activities other than work, in particular my family, friends with whom I play music, and hiking and orienteering companions. I did not cite all of you who helped me conducting this work, but I send you heartfelt thanks.

Below is the list of all my co-authors. I warmly thank all of them for the pleasure of sharing some exciting research work.

- Kunal Agrawal, MIT, USA
- Marco Aldinucci, University of Torino, Italy
- Leonardo Brenner, LIG, Grenoble, France
- Henri Casanova, University of Hawai'i, USA
- John P. Chick, University of Edinburgh, UK
- Murray Cole, University of Edinburgh, UK
- Alexandru Dobrila, University of Franche-Comté, France
- Jan Duennweber, University of Muenster, Germany
- Fanny Dufossé, LIP, ENS Lyon, France
- Paulo Fernandes, PUCRS, Porto Alegre, Brazil
- Matthieu Gallet, LIP, ENS Lyon, France

- Bruno Gaujal, LIG, Grenoble, France
- Stephen Gilmore, University of Edinburgh, UK
- Sergei Gorlatch, University of Muenster, Germany
- Yi Gu, University of Memphis, USA
- Mourad Hakem, IUT Belfort, France
- Jane Hillston, University of Edinburgh, UK
- Harald Kosch, Univerity Passau, Germany
- Loris Marchal, LIP, ENS Lyon, France
- Jean-Marc Nicod, University of Franche-Comté, France
- Laurent Philippe, University of Franche-Comté, France
- Brigitte Plateau, LIG, Grenoble, France
- Jean-François Pineau, LIP, ENS Lyon, France
- Veronika Rehn-Sonigo, LIP, ENS Lyon, France
- Yves Robert, LIP, ENS Lyon, France
- Arnold Rosenberg, Colorado State University, USA
- William J. Stewart, NCSU, Raleigh, USA
- Eric Thierry, LIP, ENS Lyon, France
- Frédéric Vivien, LIP, ENS Lyon, France
- Qishi Wu, University of Memphis, USA

CHAPTER 1

Introduction

This *habilitation* thesis is targeting the study of pipelined applications. Our goal is to schedule such applications onto large-scale distributed platforms, in order to optimize one criterion or several criteria.

In Section I, we explore the context of this work and precisely define what *scheduling* and *pipelined applications* mean, together with the *optimization criteria.* Then we give an overview of the document in Section II, and we detail in which context (and in collaboration with whom) each piece of the work presented in this document has been conducted.

I Context

I.1 Scheduling for large-scale distributed platforms

The problem of *scheduling* can be described as follows: where and when should computations be executed? This implies that we have an application, consisting of a set of computations, which needs to be executed. The execution must be orchestrated on a computational platform, which consists of a set of heterogeneous distributed resources. The problem turns out to be a *mapping* problem if we just need to answer the question “where”, i.e., to decide which part of the application runs on which resource. Since resources are distributed, the different parts of the applications may need to exchange data, and particular care must be taken on the communication model.

The mapping or scheduling problem is in fact an optimization problem in which some criteria must be optimized. Classical scheduling for parallel machines aims at mapping a task graph, i.e., an application made of several tasks with dependencies, onto a set of distributed resources. The optimization criterion is then the *makespan*, i.e., how can we map the different tasks onto processors so as to minimize the execution time of the application? It is well known that even simple instances of this classical optimization problem, on a homogeneous execution platform, are already NP-hard [27]. Several scheduling and load-balancing techniques have been developed for homogeneous architectures (see [110] for a survey) but the advent of heterogeneous clusters and grids [50, 47] has rendered the mapping problem even more difficult. Several heuristics have been introduced to schedule task graphs on different-speed processors, see [94, 120] among others. Unfortunately, such heuristics often assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications.

In contrast with traditional scheduling, we target different applications, different platforms, and thus naturally different optimization criteria:

- Rather than scheduling task graphs, we consider mostly *structured pipelined applications*, i.e., a simpler task graph which is executed many times with different input data (see Section I.2).

- Rather than targeting a homogeneous platform with an unrealistic communication model, we aim at tackling *heterogeneous dynamic platforms* which take communication contention into account, and which may be subject to unrecoverable interruptions (see Section I.3).

- Rather than focusing on makespan minimization, we consider new optimization criteria which naturally emerge from the new applications and the new platforms. With many different and often contradictory objectives, trade-offs must be taken, and we need to define a *multi-criteria objective function* as our scheduling goal (see Section I.4).

I.2 Applications: structured parallel programming

Structured parallel programming approaches have been introduced in order to rule out many of the problems which the low-level parallel application developer is usually confronted by, such as deadlocks or process starvation. These problems are particularly present when scheduling onto dynamic heterogeneous platforms, as in the context of our work. One productive approach to high-level structured parallel programming is to use *algorithmic skeletons* [39, 106, 41] to structure the creation and configuration of processes. In this approach, the skeletons add expressive power to the programming language used for sequential computing blocks, and expedite the development of complex parallel applications by providing generic and parametric parallel processing constructs to complement the loop constructs and conditional statements which are used in sequential computation.

In such a setting, typical applications continuously operate on a stream of data sets, hence the term *pipelined applications*. An application is partitioned into tasks, or stages, that are linked by simple constraints, such as for instance a linear chain of precedence or a fork graph. In the most general case, constraints are described with a directed acyclic graph (usually called DAG). In steady-state, data sets are pumped from one task to its successor. The problem is to map tasks onto resources and to organize the schedule of communications and computations so that no bottleneck happens to slow down the entire process. It is a difficult problem to decide which (and how many) resources to use for mapping each task.

Rather than being expressed as a general unstructured DAG, many important applications fit within a range of well-known solution paradigms, such as linear or fork-join computations. High-level approaches based on algorithmic skeletons identify such patterns and seek to make it easy for an application developer to tailor such a paradigm to a specific problem. A library of skeletons is provided to the programmer, who can rely on these already coded patterns to express the communication scheme within her/his own application, see for instance the eSkel library [C7],[C12]. Moreover, the use of a particular skeleton carries with it considerable information about implied scheduling dependencies, which we believe can help to tackle the complex problem of scheduling a distributed application onto a heterogeneous platform.

Finally, we point out that algorithmic skeletons address the challenges of grid computation well. In response to changing workload on servers, or unplanned unavailability due to software or hardware faults, the application can be restructured to use an alternative implementation skeleton or can simply reevaluate the parameters to the skeleton which is currently in use. Such resilience to operational faults is not typically found in low-level parallel programming approaches and is one of the strengths of a structured approach to parallel programming. Furthermore, the use of skeletons allows the programmer to provide explicit information about the future interaction structure of the application, which would be difficult or impossible to derive statically from an equivalent unstructured program source.

In the major part of this work, we focus on pipelined applications that can be expressed as algorithmic skeletons, and in particular the so-called *pipeline skeleton*, i.e., applications whose dependence graph is a linear chain. Indeed, this particular skeleton is one of the most widely used. In such applications, a series of data sets enter the input stage and progress from stage to stage until the final result is computed [115, 116, 112, 123]. Each stage has its own communication and computation requirements: it reads an input from the previous stage, processes the data and outputs a result to the next stage. For each data set, initial data is input to the first stage, and final results are output from the last stage. Such applications are a popular programming paradigm for streaming applications like video and audio encoding and decoding, DSP applications, etc [45, 118, 129]. The application operates in synchronous mode: after some latency due to the initialization delay, a new data set is completed every period. Informally, the period can often be defined as the longest cycle-time to operate a stage, and it is the inverse of the throughput that can be achieved. On the other hand, the latency is the maximum time required to compute one single data set entirely (i.e., the equivalent of the makespan for pipelined applications). Both these performance criteria are important optimization functions for such streaming applications.

Note that most scheduling problems for pipelined applications turn out to be NP-hard if the application is not structured, see for instance [118, 18]. In [18] the problem becomes polynomial for special classes of DAGs, such as series-parallel graphs. Similarly, in the DataCutter project [45], task graphs are more general than linear chains or forks, but still more regular than arbitrary DAGs, which makes it possible to design efficient heuristics to solve the optimization problems. This illustrates once again the utility of a structured approach.

I.3 Platforms: communication model, variability, failures

We first discuss communication models for the heterogeneous distributed platforms that we target, before introducing dynamic platforms. We conclude the section with a brief discussion on alternative platforms.

Distributed platforms and communication models. Distributed-memory parallel computing platforms pose many challenges to the algorithm designer and the programmer. An obvious factor contributing to this complexity is the need for network communication, whose performance is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms. In light of the complexity of performance modeling for network communications, the vast majority of scheduling works and results address a very simple model which assumes that there is no contention for network links. In other words, a processor can send distinct messages to a thousand of processors at the same speed as if there were a single message!

Recent papers [63, 67, 111] suggest to take communication contention into account. Among these extensions, scheduling heuristics are considered in [16], in which each processor can communicate with at most one other processor at a given time-step (one-port model, [73, 85, 23]). In Chapter 2, we extensively discuss various realistic communication models on which we conduct our study.

Dynamic platforms are characterized by their larger size, and greater degree of heterogeneity. The resources of these platforms (topology, message routes, etc.) and their characteristics are assumed to be known by a centralized control mechanism, even though they change over time. Radical changes are caused by failures, but the performance of a processor can also be slightly reduced because, for instance, another user launched a new process onto this resource. A typical example of such platform is a general purpose computational grid [51], or a set of resources provided by a team of users that can change significantly over time, such as in volunteer computing [83]. Scheduling techniques which aim at dealing with the dynamic nature of such platforms are extensively discussed in Chapter 4.

Note that a scheduling algorithm for such platforms may aim at optimizing the *reliability* of the schedule, in addition to the usual performance criteria that have been discussed so far, such as the application throughput and the latency (or makespan).

Alternative platforms. The scheduling of pipelined computations can also be conducted on special-purpose architectures and FPGA arrays, see for instance the representative work by Fabiani and Lavenier [48]. They study the placement of linear computations onto reconfigurable arrays. Another line of work is related to the design of fault-tolerant or power-aware mapppings for embedded systems. Representative examples are [133, 8]. We do not target such special-purpose architectures, but rather limit our study to large-scale distributed platforms.

I.4 Objective functions: multi-criteria optimization

Even though optimization criteria will be extensively discussed in Chapter 2, we have already seen that, in addition to the classical "makespan" objective, our framework raises many new questions. Can we design a schedule which ensures a good throughput of the application? Can we design a robust schedule, which can support failures and platform variability? How do we mix such different objective functions?

In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But is it natural for the user to maximize the quantity $0.7T + 0.3R$, where T is the throughput and R the reliability? Obviously, the problem here is that we mix apples and bananas: the criteria are very different in nature and it does not make much sense for a user to make a linear combination of them. Users are more likely to ask questions like "I want a throughput T, how reliable can my application be?" In our work, we advocate the use of multi-criteria with thresholds: one single criterion is optimized, under the condition that a threshold is enforced for all other criteria.

II Overview of the document

II.1 Linear chain pipelined applications

Chapters 2 and 3 focus on the problem of scheduling linear chain pipelined applications onto heterogeneous platforms, with multi-criteria objective functions, as discussed in Section I. Their goal is to offer a good view of the state of the art research on such problems. It is a synthesis of several pieces of work that were conducted with various colleagues in the last few years.

Chapter 2 accurately defines the models and the scheduling problems. Many surprising problems have been raised recently, in particular the difficulty to compute the throughput and the latency of a given mapping, especially in a bi-criteria setting. With less surprise, we illustrate the fact that the choice of the communication model may have a dramatic impact on the encountered difficulties.

The complexity results are described in Chapter 3. Of course we omit in this chapter many technical proofs, which are very involved (but references are given for the interested reader). For instance, it took us several months to figure out how to prove that the latency problem is NP-hard when restricting to interval mappings on fully heterogeneous platforms, and we needed to discuss with an expert in graph theory, Eric Thierry, in order to find the key idea.

Historically, the first study that we conducted was the throughput maximization under the one-port model with no overlap, together with Yves Robert (see [J9]). Then, after a visit to the group of Umit Catalyurek and Joel Saltz (Colombus, Ohio), we realized that it would be interesting to study trade-offs between throughput and latency. This lead to a theoretical bi-criteria study (see [J10]). Our PhD student Veronika Rehn-Sonigo got involved in the subject, and we worked with her both on the reliability criterion and on a more experimental study based on a JPEG encoder application, in collaboration with Harald Kosch (see for instance [C19], [J12]). We also started investigating a model with overlap while visiting Kunal Agrawal at MIT, USA (see [C24]). The full understanding of the correct model for period and latency definition is much more recent, and some NP-completeness proofs are not yet published at the time this document is written. Most of this part of the work was strongly inspired by our work with Kunal Agrawal and Fanny Dufossé on the mapping of filtering services rather than linear chain applications, see [C32].

II.2 Dynamic platforms and complex applications

In Chapter 4, we exhibit the difficulties encountered when dealing with dynamic platforms, and we show how performance models can help us with the study.

The earlier work on this subject was conducted during my post-doctoral position in Edinburgh, with Murray Cole, Stephen Gilmore and Jane Hillston, where we developed a first performance model based on performance evaluation process algebra PEPA [62], see [J3],[J4]. Some related projects such as the ICENI project [53] also used performance models to improve the scheduling decisions, but these are just graphs which approximate data obtained experimentally. Moreover, there is no upper-level layer based on skeletons.

Building upon the weaknesses of the previous model, and using the expertise of Bruno Gaujal on timed Petri nets [11], we designed with Matthieu Gallet and Yves Robert a new model. This helped us capture the difficulty of period definition, since we realized that there are cases in which there are no critical resources (see [C34]).

The third part of the chapter is dedicated to a simpler application, consisting in a divisible workload to be scheduled on a homogeneous platform, but with a complex probabilistic model for failures. This work was initiated while visiting Arnold Rosenberg in his wonderful house at Falmouth (Cape Cod), together with Yves Robert and Frédéric Vivien. It turned out to be surprisingly difficult and we spent several weeks before we were able to define a convincing failure model (see the result in [C25]).

Chapter 5 is also a collection of various work, but it rather tackles general applications.

We first consider filtering applications, whose study turns out to be very close to the one conducted for linear chain pipelined applications. I worked on this subject with Yves Robert and Fanny Dufossé, during Fanny's master and PhD thesis (see [C26],[C29]). We pursued this study with Kunal Agrawal, and it was through the study of filters that we got a clear vision of the difficulties of period computation and the impact of the communication model, see [C32].

Naturally, we also decided to tackle applications with more complex dependencies than linear chains, such as the study on fork and fork-join graphs conducted with Yves Robert in [J10]. For general DAGs, all problems turn out to be NP-complete, but we designed fault tolerant heuristics with Mourad Hakem, see [J11],[J15].

The two last applications were studied during the PhD thesis of Veronika Rehn-Sonigo, which I supervised together with Yves Robert, and who worked on multi-criteria optimization problems. The replica placement problem described in this chapter is an interesting application, and surprisingly, we could find only very constrained placement policies in the literature. Therefore, we decided to introduce and study new policies (see [J8]). We did not work on all the possible extensions suggested in the literature since problems rapidly became too complex, but already we gave a good overview of the difficulties of replica placement.

Finally, the work on in-network stream processing was conducted together with Henri Casanova and initiated during a marvelous trip to Hawai'i. This complex application allowed us to design sophisticated heuristics and linear programs, and it illustrated the combined difficulties of all applications (see [C28],[C37]).

II.3 Conclusion and appendices

A general conclusion on the work presented in this document, together with a description of current and future work, is available in Chapter 6. I did not summarize all my past research work in this document, but focused on the work related to my main research activity, namely the study of the scheduling of pipelined applications. In particular, there is no mention of the research work done during my PhD thesis.

In Appendix A, one can find the exhaustive list of my publications, including the work done during my PhD thesis and my post-doctoral research position. A large set of references is also provided, even though some detailed discussions on related work have been omitted in this document (but are available in my publications corresponding to each piece of work).

Finally, a brief curriculum vitae can be found in Appendix B.

CHAPTER

Linear chain application models 2

This chapter is devoted to a precise definition of models for the scheduling of linear chain pipelined applications. We first focus on the linear chain pipelined application model in Section I. Then we describe in Section II the computational platform on which we aim at scheduling the application, and we detail several communication models, since these models have a big impact on the problem complexity. Section III presents the definition of different application types, mapping rules, and various objective criteria, which allows us to formally define a set of optimization problems. Finally, we summarize the difficulties encountered during the modeling process of linear chain applications and we conclude in Section IV. Note that all notations used throughout this chapter are summarized in Table 2.1 (page 33).

I Linear chain pipelined applications

As already stated in Chapter 1, we focus in this work on pipelined applications, i.e., applications which continuously operate on a stream of data sets. Moreover, many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic algorithmic patterns. A skeleton [39, 106, 41] is a programming construct which abstracts such a pattern of processes and interactions. The programmer invokes one or more skeletons to describe the structure of a program, specializing each with types and operations from the application domain. Code handling the interaction and invocation of the domain specific operations is inherited implicitly from the chosen skeleton.

In the simplest form of pipeline parallelism [40], a sequence of n stages process a sequence of *inputs* to produce a sequence of *outputs* (Figure 2.1). All inputs pass through each stage in the same order, with the processing of a particular input beginning as soon as its predecessor has left the first stage. Note that parallelism is introduced by overlapping the processing of many input instances (i.e., data sets). It is then quite normal for the processing time of each data set to be increased by pipelining. However, performance benefits (in terms of throughput) accrue when many data sets are processed concurrently across the pipeline. Such linear graphs

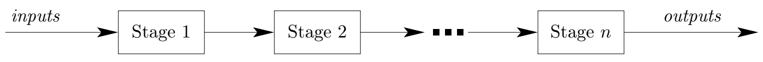


Figure 2.1: Linear chain application.

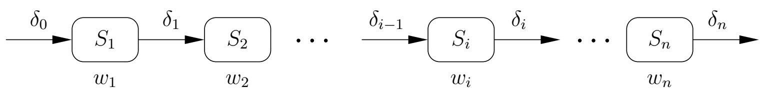


Figure 2.2: Linear chain application with parameters.

are representative of a wide class of applications, and constitute the typical building blocks upon which to build and execute more complex applications.

Formally, a linear chain application consists of n stages S_i, $1 \leq i \leq n$. Consecutive data sets are fed into the pipeline and processed from stage to stage, until they exit the pipeline after the last stage. Each stage executes a task. More precisely, the i-th stage S_i receives an input from the previous stage, of size δ_{i-1}, performs a number of w_i computations, and outputs data of size δ_i to the next stage. This operation corresponds to the i-th stage and is repeated periodically on each data set. The first stage S_1 receives an input of size δ_0 from the outside world, while the last stage S_n returns the result, of size δ_n, to the outside world. An example of application with all its parameters is given on Figure 2.2.

II Target platforms and communication models

The aim of this work consists in executing a linear chain application on a target platform, so as to optimize some performance criteria, and this section is devoted to platform models. We first propose a general model of the target platform in Section II.1, and we focus on several platform types which are important in II.2. Then we discuss the case in which processors are subject to failure (Section II.3), and finally we describe various communication models in Section II.4.

II.1 Target platform

We target a heterogeneous platform with $p + 2$ processors P_u, $0 \leq u \leq p + 1$, as illustrated in Figure 2.3. $P_{in} = P_0$ and $P_{out} = P_{p+1}$ are two special additional processors devoted to input/output data: initially, the input data for each task resides on P_{in}, while all results must be returned to and stored in P_{out}. These special processors are thus connected to all other processors $P_u, 1 \leq u \leq p$. Processors P_u, with $1 \leq u \leq p$, are fully interconnected as a (virtual) clique. Therefore, there is a bidirectional link $link_{u,v} : P_u \leftrightarrow P_v$ between any processor pair P_u and P_v, with $0 \leq u \leq p$ and $1 \leq v \leq p+1$, of bandwidth $b_{u,v}$. Note that we do not need to have a physical link between any processor pair. Instead, we may have a switch, or even a path composed of several physical links, to interconnect P_u and P_v; in the latter case we would retain the bandwidth of the slowest link in the path for the value of $b_{u,v}$.

In addition to link bandwidths, we have in some cases processor network cards that bound the total communication capacity of each computing resource. We denote by B^i_u (resp. B^o_u) the capacity of the input (resp. output) network card of proces-

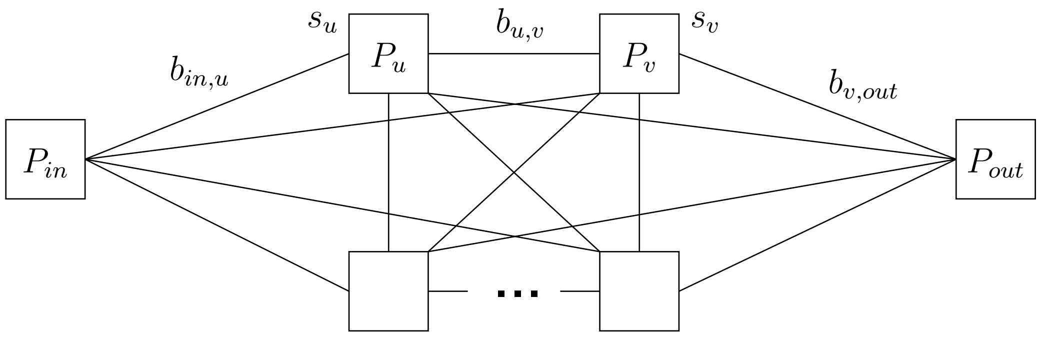


Figure 2.3: The target platform.

sor P_u. In other words, P_u cannot receive more than B^i_u data items per time-unit, and it cannot send more than B^o_u data items per time-unit.

In the most general case, we have fully heterogeneous platforms, with different processors speeds and link capacities. The speed of processor P_u is denoted as s_u, and it takes X/s_u time-units for P_u to execute X floating point operations. We also enforce a linear cost model for communications, hence it takes $X/b_{u,v}$ time-units to send (resp. receive) a message of size X to (resp. from) P_v.

II.2 Platform classification

We classify below particular platform cases which are important, both from a theoretical and practical perspective:

Fully Homogeneous – These platforms have identical processors ($s_u = s$) and homogeneous communication devices ($b_{u,v} = b$ for link bandwidths, and $B^i_u = B^i$, $B^o_u = B^o$ for network cards). They represent typical parallel machines.

Communication Homogeneous – These platforms are still interconnected with homogeneous communication devices, but they have different-speed processors ($s_u \neq s_v$). They correspond to networks of workstations with plain TCP/IP interconnects or other LANs.

Fully Heterogeneous – These are the most general, fully heterogeneous architectures. Hierarchical platforms made up with several clusters interconnected by slower backbone links can be modeled this way.

II.3 Unreliable processors

In the previous classification of platforms, heterogeneity may come from either processor speed or link bandwidth. Yet another source of heterogeneity is the reliability of each target processor. To model this, we associate a failure probability $0 \leq f_u \leq 1$, $1 \leq u \leq p$ to each processor. It is the probability that the processor breaks down during the execution of the application. We consider constant failure probabilities as we are dealing with pipelined applications. These applications are meant to run during a very long time, and therefore we address the question of whether the processor will break down or not at any time during execution. It might seem odd that this probability is completely independent of the execution time, since one may believe that the longer a processor executes, the larger the chance that it fails. However, we target a steady-state execution of the application, for instance in a scenario in which we would loan/rent resources. Computers could be suddenly reclaimed by their owners, as during an episode of *cycle-stealing* [7, 24, 107]. The failure probability should thus be seen as a global indicator of the reliability of a processor. Also note that we consider in this work only fail-silent (a faulty processor does not produce any output) and fail-stop (no processor recovery) processor failures. We do not consider link failures since in a grid framework, a different path can be found to tackle such failures. However, we do target dynamic platforms in Chapter 4, and consider a dynamic behavior from both links and processors.

A platform composed of processors with identical failure probabilities is denoted *Failure Homogeneous* and otherwise *Failure Heterogeneous*. Note that it seems natural to consider *Failure Heterogeneous* platforms when processors have different speeds, thus for *Communication Homogeneous* or *Fully Heterogeneous* platforms, while *Fully Homogeneous* platforms are more likely to be *Failure Homogeneous*.

II.4 Communication models

Distributed-memory parallel computing platforms pose many challenges to the algorithm designer and to the programmer. An obvious factor contributing to this complexity is the need for network communication, whose performance is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms. In light of the complexity of performance modeling for network communications, the vast majority of scheduling works and results are for a very simple model, which is as follows [130, 86, 121]. If a task T communicates data to a successor task T', the cost is modeled as

$$cost(T,T') = \begin{cases} 0 & \text{if } alloc(T) = alloc(T') \\ comm(T,T') & \text{otherwise,} \end{cases}$$

where $alloc(T)$ denotes the processor that executes task T, and $comm(T,T')$ is defined by the application specification. The above model states that the time for communication between two tasks running on the same processor is negligible. The model also assumes that the processors are part of a fully connected clique. This so-called *macro-dataflow* model makes two main assumptions: (i) communication can occur as soon as data are available; and (ii) there is no contention for network links. Assumption (i) is reasonable as communications can overlap with computations in most modern computers. Assumption (ii) is much more questionable. Indeed, there is no physical device capable of sending, say, 100 messages to 100 distinct processors, at the same speed as if there were a single message. In the worst case, it would take 100 times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources! Nevertheless, this assumption is omnipresent in the traditional scheduling literature. Perhaps it was the price to pay to derive tractable mathematical results on makespan minimization?

Our conviction is that we need to turn to more realistic communication models when modeling concurrent communications, in order to obtain a much better trade-off between realism and tractability. We outline two such models, that account for the interference between concurrent communications.

II.4.1 The one-port model without overlap

A radical option is simply to forbid concurrent communications at each node. In the *one-port* model [22, 23], a given processor can be involved in a single communication at any time-step, either a send or a receive. However, independent communications between distinct processor pairs can take place simultaneously. This model is thus very pessimistic as real-world platforms can achieve some concurrency of communication. On the other hand, it is straightforward to design algorithms that follow this model and thus to determine their performance a priori.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes simpler models [14, 88, 81] where communication time only depends on the sender, not on the receiver. In these models, the communication speed from a processor to all its neighbors is the same.

It is used by Bhat et al. [22, 23] for fixed-sized messages. They advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has been related by Saif and Parashar [108], who report that asynchronous sends become serialized as soon as message sizes exceed a few tens of kilobytes[1].

Another key assumption to define the execution model is to decide whether computation can overlap with (independent) communication. Since the one-port model serializes communications, it is natural to assume that communication and computation also are serialized, and thus we consider in this case a model with no overlap.

II.4.2 The bounded multi-port model with overlap

Assuming an application that runs threads on, say, a node that uses multicore technology, the network link could be shared by several incoming and outgoing communications. Therefore, the sum of the bandwidths allotted by the operating system to all communications cannot exceed the bandwidth of the network card. The

[1]Their results hold for two popular implementations of the MPI message-passing standard, MPICH on Linux clusters and IBM MPI on the SP2.

bounded multi-port model proposed by Hong and Prasanna [64, 66] assesses that an unbounded number of communications can thus take place simultaneously, provided that they share the total available bandwidth. We point out that recent multi-threaded communication libraries[2] now allow for initiating multiple concurrent send and receive operations, thereby providing practical realizations of the multi-port model.

Note that with this model there is no degradation of the aggregate throughput. Such a behavior is typical for protocols with efficient congestion control mechanisms (e.g., TCP). Note, however, that this model does not express how the bandwidth is shared among the concurrent communications. It is generally assumed in this model that the application is allowed to define the bandwidth allotted to each communication. In other words, bandwidth sharing is performed by the application and not by the operating system. While technology exists to achieve application-level bandwidth sharing, it is not the standard way in which networks and operating systems operate (yet?).

On homogeneous platforms it would be implemented with one-port communications, because if the links have same bandwidths it is better to send messages serially than simultaneously. However, the bounded multi-port is more flexible for heterogeneous platforms.

In this bounded multi-port model, it is natural to assume full overlap of communications and computations, so that a server can receive, compute and send (independent) data simultaneously. Indeed, most state-of-the-art processors running a threaded operating system are capable of such an overlap. These two assumptions (multi-port and overlap) thus fit well together because they both require a multi-threaded system.

II.4.3 Other communication models

There are more complicated models such as those that deal with bandwidth sharing protocols [96, 92]. Such models are very interesting for performance evaluation purposes, but they almost always prove too complicated for algorithm design purposes. For this reason, we prefer to deal with the one-port or the bounded multi-port model. As stated above, we believe that these models represent a good trade-off between realism and tractability.

[2]See for instance MPICH2 [78].

III Multi-criteria mapping problems

This section is devoted to the definition of the multi-criteria mapping problems. We start by defining different stage types and replication mechanisms, and by establishing different mapping rules in Section III.1. Then we introduce and motivate several optimization criteria and we explain how to simultaneously deal with multiple criteria in Section III.2. Finally, we define and classify the various optimization problems under study in Section III.3.

III.1 Stage types and mapping rules

The general mapping problem consists in assigning application stages to platform processors. The properties of application stages can differ, depending upon the application, and thus we first introduce the different types of stages, and discuss replication mechanisms. Then we define the allocation function which assigns stages to processors; some constraints can be added to the mapping to ease the implementation of the application, and different instances of the mapping problem are discussed.

III.1.1 Stage classification

In this work, three types of stages are considered: monolithic, dealable and data-parallel stages. We also explain the replication mechanism for reliability, since it looks similar to the other replication techniques, while being truly different.

Monolithic stages – If stage S_i is a sequential procedure which may perform disc accesses or write data in the memory for each data set, S_i is said to be *monolithic*. Indeed, this data may be reused from one data set to another, and thus the rule of the game is always to process the data sets in a sequential order within a monolithic stage. Moreover, due to the possible local memory accesses, S_i must be mapped onto a single processor: we cannot process half of the data sets on a processor and the remaining ones on another without exchanging intra-stage information, which might be costly and difficult to implement.

Dealable stages – If the computations of stage S_i are independent from one data set to another, S_i can be replicated as a deal [115, 116, 112, 41, 20, 124]: several consecutive computations are mapped onto distinct processors, and data sets are processed in a round-robin fashion[3] by these processors (hence the term *deal*). The computations of a dealable stage can be fully sequential for a given data set, as long as they do not depend from previous results for other data sets.

[3]This round-robin behavior will be further motivated in the next section.

Data-parallel stages – If the computations of stage S_i are data-parallel, their execution can be split among several processors. Contrarily to replicated stages where different instances (for different data sets) are assigned to different resources, each instance of a data-parallel stage is assigned to several processors, which speeds-up the production of each result. There is another major difference: while we can replicate intervals of consecutive stages, we can only data-parallelize single stages (but maybe several of them). To see why, consider two consecutive stages, the first one executing some low-level filtering on its input file (an image), and the second stage implementing various high-level component extraction algorithms. Both stages can be made data-parallel, but the entire image produced by the first stage is needed as input to the second stage. In fact, if both stages could have been data-parallelized simultaneously, the application designer may have chosen to gather them into a single stage, thereby giving more opportunities for an efficient parallelization.

Replicating for failures – In order to handle failures, another replication mechanism can be used, for any stage type. The idea consists in doing redundant work: several processors will process the same stage on the same data sets. Thus in case of failure of one of the replicated processors, we still get a result. This kind of replication is different from deal or data-parallel replication, and both kinds of replication can be mixed.

Note that we use the term *replication* for several meanings. However, only the replication for reliability is really a replication of the work, while deal replication is rather a *distribution* of the work, and data-parallel replication is rather a *partitioning* of the work. However, to be consistent with the literature, we keep the term *replication* in the following, and we always specify if we consider replication for performance (deal or data-parallelism), or replication for reliability.

III.1.2 Allocation function and mapping rules

We aim at assigning stages onto processors, and this is achieved with an allocation function. Two fictitious stages S_0 and S_{n+1} are created, and allocated respectively to P_{in} and P_{out}. We first assume that all stages are monolithic and that no replication is done for reliability, in order to define different mapping rules in a simpler context. Then we extend definitions to scenarios with replication.

Simple scenario with no replication. In this first scenario, the allocation function $alloc : [1..n] \rightarrow [1..p]$ associates one single processor index to each stage index. This function is also extended for the fictitious stages: $alloc(0) = 0$ $(= in)$ and $alloc(n+1) = p+1$ $(= out)$.

The first mapping rule restricts the search to One-to-one mappings: each stage is mapped onto a *distinct* processor, and thus *alloc* is a one-to-one function. Note that such a mapping is possible only if $n \leq p$.

One-to-one mappings may be needlessly restrictive. Natural extensions are Interval mappings in which each participating processor is assigned an interval of consecutive stages. Note that when $p < n$ interval mappings are mandatory. Intuitively, assigning several consecutive tasks to the same processors will increase its computational load, but will also decrease communication. The best interval mapping may turn out to be a one-to-one mapping, or instead may utilize only a very small number of fast computing processors interconnected by high-speed links. An interval mapping is defined with a partition of $[1..n]$ into $m \leq p$ intervals $I_j = [d_j, e_j]$ such that $d_1 = 1$, $d_j \leq e_j$ for $1 \leq j \leq m$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m-1$ and $e_m = n$. Recall that the function *alloc* associates a processor index to each stage index. In a one-to-one mapping, this function was a one-to-one assignment. In an interval mapping, for $1 \leq j \leq m$, the whole interval I_j is mapped onto the same processor $P_{alloc(d_j)}$, i.e., for $d_j \leq i \leq e_j$, $alloc(i) = alloc(d_j)$. Also, two intervals cannot be mapped to the same processor, i.e., for $1 \leq j, j' \leq m$, $j \neq j'$, $alloc(d_j) \neq alloc(d_{j'})$.

The most general mappings may be more complicated than interval mappings. In a General mapping, a processor P_u can be assigned any subset of stages. Thus, there are no constraints on the allocation function.

Extension to replication. These definitions can easily be extended to replication: instead of returning a single processor index, the allocation function needs to return a set of processor indices. We also need to specify which of those processors are used to replicate dealable stages, for data-parallelism, and also which processors are used for reliability replication. For a stage index $1 \leq i \leq n$, we denote by $alloc(i)$ the whole set of processor indices. This set is partitioned into t_i *teams*, and each team consists of one or more processors. Each processor within a team is allocated the same piece of work, and thus inside a team, redundant work is done for reliability issues. Teams for stage S_i are denoted as $T_{i,1}, \ldots, T_{i,t_i}$.

- If stage S_i is neither dealable nor data-parallel, then all processors work on the same data sets, and thus they form a single team: $t_i = 1$ and $|T_{i,1}| = |alloc(i)|$. In this case, if $|alloc(i)| > 1$, the replication only helps increasing the reliability of the application.
- If data sets are distributed in round-robin (dealable stage), each team is in charge of one round of the deal. We set $type_i = deal$ to differentiate it from a data-parallel execution of a stage.
- If the stage is data-parallel, $type_i = dp$. In this case, each team is in charge of the computation of a fraction of each data set. The computation of one data set is thus executed in parallel on t_i processors, one in each team.

Of course, the three mapping rules can still be defined when there is replication.

- One-to-one mappings are such that $alloc(i) \cap alloc(i') = \emptyset$ for $i \neq i'$.
- Interval mappings are such that $alloc(i) = alloc(d_j)$, $t_i = t_{d_j}$ and $type_i = type_{d_j}$ for each stage i in interval j, and $alloc(d_j) \cap alloc(d_{j'}) = \emptyset$ for $j \neq j'$. Also, recall that only single stages can be data-parallelized (and not intervals), thus we enforce that $type_{d_j} = dp \Rightarrow d_j = e_j$.

- For General mappings, we enforce that a processor can be involved only in a single team. Thus we keep a notion of *interval*, or more precisely *subset* of stages, where d_j is the index of the first stage of subset j, and $stages_j$ the set of indices of stages in this subset. For i in $stages_j$, we still enforce that $alloc(i) = alloc(d_j)$, $t_i = t_{d_j}$ and $type_i = type_{d_j}$. The difference is that subsets are not made of consecutive stages as before. Also, we keep the constraint stating the team exclusion in terms of processors: $alloc(d_j) \cap alloc(d_{j'}) = \emptyset$ for $j \neq j'$. Finally, the data-parallel constraint for single stages now writes: $type_{d_j} = dp \Rightarrow |stages_j| = 1$.

Note that we could envision fully general mappings with no constraints, but our definition of General mappings makes good sense, and it is already the source of several difficulties, so we did not tackle fully general mappings yet.

We conclude this section with a comment on the round-robin rule enforced for the mapping of dealable stages onto several processors. With different speed processors, a more efficient strategy to replicate a stage interval would be to let each processor execute a number of instances proportional to its speed. For instance, with one fast processor of speed 2 and a slower one of speed 1, we could assign twice as many data sets to the fast processor than to the slow one. Each resource would then be fully utilized. However, such a demand-driven assignment is quite likely to lead to an out-of-order execution of data sets in the general case: because of the different pace at which processors are executing the computations, the k-th data set may well exit the replicated stage interval later than the $k+1$-st data set. This would violate the semantics of the application if, say, the next stage is monolithic. Because in real-life applications, some stages are monolithic and some can be replicated, the round-robin rule is always enforced [41, 106].

III.2 Optimization criteria

Before defining the optimization problem, we need to identify objective functions. In fact, our aim is to find the best allocation function, i.e., the allocation function which optimizes one (or several) criterion (criteria). In this section we first describe each criterion separately (Section III.2.1), and then we explain how to simultaneously deal with multiple criteria in Section III.2.2. Finally, we give a formal definition of period and latency in Section III.2.3.

III.2.1 Criteria definition

For pipelined applications, the first objective that comes to mind is *throughput* maximization: the goal is to process as many data sets per time unit as possible. The throughput measures the aggregate rate of processing of data, and it is the rate at which data sets can enter the system. Equivalently, the inverse of the throughput, defined as the period, is the time interval required between the beginning of the execution of two consecutive data sets. For each data set, a processor reads the input from its predecessor, executes computations corresponding to all the stages assigned to it, and sends the output to its successor. The processor periodically repeats this operation cycle every period[4].

However, looking back at classical scheduling, makespan minimization was an important objective too. This remains true for pipelined applications, and in particular for real-time applications. The definition must be adapted, and we talk of *latency* rather than of makespan, in order to avoid confusion. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. Note that it may well be the case that different data sets have different latencies (because they are mapped onto different processor sets), hence the latency is defined as the maximum response time over all data sets.

We can already note that minimizing the latency is antagonistic to maximizing the throughput. In fact, intuitively, assigning all application stages to the fastest processor (thus working in a fully sequential way) would suppress all communications and accelerate computations, thereby minimizing the latency, but achieving a very bad throughput. Conversely, mapping each stage to a different processor is likely to decrease the period, hence increase the throughput (work in a fully pipelined manner), but the resulting latency will be high, because all inter-stage communications must be accounted for in this latter mapping. Already we guess that trade-offs will have to be found between these criteria. Indeed, several works deal with both these criteria, for instance see [116],[J10].

With the advent of large-scale heterogeneous platforms, resources may be cheap and abundant, but resource failures (processors/links) are more likely to occur and have an adverse effect on the applications. Not only every user is quite likely to face unrecoverable hardware failures when deploying applications on clusters or grids [52, 55, 2, 46], but unrecoverable interruptions can also take place in other important frameworks, such as loaned/rented computers being suddenly reclaimed by their owners, as during an episode of *cycle-stealing* [7, 24, 107]. What if some processor speed suddenly decreases (or worse the processor is no longer responding)? The throughput and latency would be severely impacted by this sudden bottleneck. Obviously, the mapping must be robust: it should be prepared, so to speak, to react to resource variations, and should be capable to reserve more resources than needed so as to re-allocate computations as the execution progresses, based on some histogram of the current execution. A solution consists in replicating the processing of key stages, mapping them onto distinct sets of resources, and gathering results in a data-flow operation mode. The question of determining which stages to replicate, and to which extent, raises a clear trade-off between the price to pay for robustness (or performance guarantees) and the efficient usage of resources at the system level. Altogether, there is an increasing need for developing reliable schedules. Another optimization criterion that could be maximized is thus the *reliability* of the schedule, given a failure model for the resources[5].

[4]Section III.2.3 formalizes this definition.

[5]Different failure models are investigated in Chapter 4, which targets dynamic platforms.

Another important objective emerges for current platforms, namely the *energy* minimization objective. *Green* scheduling aims at minimizing energy consumption, by running processors at lower frequencies [8], or by reducing the number of processors enrolled. Of course being "green" often involves running at a slower pace, thereby reducing the application throughput.

In addition to being green, one may also want to reduce the number of processors enrolled in order to reduce the *cost* of application execution. The user may need to pay for processing units, memory cards, network cards, or she/he may want to rent a target platform. The cost of the platform is thus another objective that can be considered, which is antagonist to the performance objectives (with fewer processors, you will not be able to be as fast and efficient).

Finally, even more objectives appear in a multi-application setting, with several concurrent applications sharing (or competing for) common computational resources. Indeed, some form of fairness must be guaranteed between all the applications. Typical measures are the maximum stretch of an application or the sum of all application stretches [19]. The stretch of an application is the slowdown factor incurred by its execution time when sharing resources with the other applications.

III.2.2 Dealing with multiple criteria

How to deal with so many objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But is it natural for the user to maximize the quantity $0.7T + 0.3R$, where T is the throughput and R the reliability? What about adding latency and energy parameters into the story? Obviously, the problem here is that we mix apples and bananas: the criteria are very different in nature and it does not make much sense for a user to make a linear combination of them.

Users are more likely to ask questions like "I want a frame rate T and a response time L for my JPEG encoder, what is the least amount of energy that I will consume?" Thus we advocate the use of multi-criteria with thresholds. To give another example, we would aim at maximizing the throughput of the application, but accepting only schedules whose reliability is at least 99%. Now, each criteria combination can be handled in a natural and meaningful way: one single criterion is optimized, under the condition that a threshold is enforced for all other criteria.

III.2.3 Formal definition of period and latency

In Section III.1.2, we have formally defined the allocation function which characterizes a mapping. However, this function alone does not give enough information to compute the actual schedule of the application, i.e., the moment at which each operation takes place. Indeed, we need the complete list of the time-steps at which each communication and computation begins and ends, in order to compute the period and the latency of the mapping. Note that we target cyclic schedules, which repeat for each data set when there is no deal replication, since each processor performs the same operations on all data sets (we explain the changes induced by deal replication later). In this case, we are able to define an operation list which is polynomial in the problem size:

- For each stage S_i $(1 \leq i \leq n)$, each processor $u \in \mathit{alloc}(i)$, and each data set k, $\mathit{BeginComp}_{i,u}^k$ is the time-step at which the computation of S_i on P_u for data set k begins, and $\mathit{EndComp}_{i,u}^k$ is the time-step at which this computation ends.
- For each stage S_i $(0 \leq i \leq n)$, each processor $u \in \mathit{alloc}(i)$ and $v \in \mathit{alloc}(i+1)$, and each data set k, $\mathit{BeginComm}_{i,u,v}^k$ is the time-step at which the communication between P_u and P_v for the output of S_i for data set k begins, and $\mathit{EndComm}_{i,u,v}^k$ is the time-step at which this communication ends.
- The schedule starts at time-step 0 with the data set number 0, and we impose a cyclic behavior of period λ:

$$\begin{cases} \mathit{BeginComp}_{i,u}^k = \mathit{BeginComp}_{i,u}^0 + \lambda \times k \\ \mathit{EndComp}_{i,u}^k = \mathit{EndComp}_{i,u}^0 + \lambda \times k \\ \mathit{BeginComm}_{i,u,v}^k = \mathit{BeginComm}_{i,u,v}^0 + \lambda \times k \\ \mathit{EndComm}_{i,u,v}^k = \mathit{EndComm}_{i,u,v}^0 + \lambda \times k \end{cases} \tag{2.1}$$

To each communication model are associated different rules that must be satisfied by the operation list so that the schedule is valid: no resource constraint nor model hypothesis is violated[6]. Note that all models are non-preemptive: once initiated, a computation or a communication cannot be interrupted. Also, communications are synchronous, and the bandwidth assigned to a given communication remains the same during its whole execution (this is not really a restriction for the one-port model but it is an important one for the multi-port model). With the operation list we can define the period and the latency of a mapping:

- The period is $\mathcal{P} = \lambda$;
- The latency is $\mathcal{L} = \max\{\mathit{EndComm}_{n,u,out}^0 \mid u \in \mathit{alloc}(n),\}$ Remember that the result for data set 0 is output to P_{out} after being processed for the last stage, S_n.

If we consider a mapping with deal replication, the definition of Equation (2.1) must be revisited since a processor may handle only part of the data sets for a given stage. For instance, if S_i is replicated on P_u and P_v, then $\mathit{BeginComp}_{i,u}^k$ is defined only for even values of k, while $\mathit{BeginComp}_{i,v}^k$ is defined for the odd values of k. We would then have $\mathit{BeginComp}_{i,u}^k = \mathit{BeginComp}_{i,u}^0 + \lambda \times k$ and $\mathit{BeginComp}_{i,v}^k = \mathit{BeginComp}_{i,v}^1 + \lambda \times (k-1)$, since P_v starts the computation for S_i on data set number 1. The same principle can be applied for communications: we define $\mathit{BeginComm}_{i,u,u'}$ and $\mathit{EndComm}_{i,u,u'}$ for the data sets which are sent

[6]An exhaustive example of such constraints can be found in [C32].

from P_u to $P_{u'}$. This can be extended to all deal replication schemes, even though it is complicated to write the general formula. Note that the mapping is not cyclic anymore, but rather periodic since different data sets are taking different execution paths. Even though the operation list is still polynomial in the problem size (at most $O(np^2)$ constraints for the beginning and end of communications), it seems difficult to check that an operation list is valid in polynomial time, since there might be an exponential number of constraints.

The period definition does not change, even though some processors in charge of a round of a deal will not process cyclically anymore, and thus have an individual period equal to λ multiplied by the number of processors involved in the deal. For the latency, however, we need to revisit the definition in order to consider the maximum time taken by any data set, and there might be an exponential number of different paths taken by data sets.

In the next section, we define the optimization problems, and it turns out that in some cases we are able to give an analytic formula to express the period and the latency of a given mapping, just based on the allocation function (and without detailing the operation list). However, in other cases in which the period is not dictated by a critical resource, finding the operation list which returns the smallest period can become a difficult challenge by itself. In such cases, the operation list which minimizes the period can either be computed with the help of a polynomial algorithm, or the problem may turn out to be NP-hard.

III.3 Optimization problems

In this section, we focus on three criteria: period, latency and reliability. We start with a study of a simple scenario with no replication, and targeting One-to-one and Interval mappings[7]. We add the concept of replication for fault tolerance, and we formally define the failure probability of an application. Then we introduce dealable and data-parallel stages: we revisit the simpler definitions and exhibit inherent difficulties of these more complex models. Finally we show the increasing difficulty introduced by General mappings.

III.3.1 Interval mappings with no replication

Given a one-to-one or an interval mapping with no dealable nor data-parallel stages (and thus no replication), recall that m processors are enrolled in the mapping (one per interval). It is then easy to define the latency: it is the maximum time required by a data set to traverse all stages. Note that only inter-processor communications need to be paid.

[7]Note that a one-to-one mapping can be expressed as an interval mapping with each interval reduced to a single stage, i.e., $m = n$ and $d_j = j$ for $1 \leq j \leq n$. Therefore we give definitions only for interval mappings.

$$\mathcal{L}^{(interval)} = \sum_{1 \leq j \leq m} \left\{ \frac{\delta_{d_j - 1}}{b_{alloc(d_j-1), alloc(d_j)}} + \frac{\sum_{i=d_j}^{e_j} w_i}{s_{alloc(d_j)}} \right\} + \frac{\delta_n}{b_{alloc(d_m), out}} \quad (2.2)$$

For the period, the definition differs depending on the communication model (the rules to be satisfied in the formal definition of Equation (2.1) are different). In all cases, the period is defined as the longest cycle-time of a processor, since the whole application will synchronize on this processor which is the bottleneck of the pipeline: $\mathcal{P}^{(interval)} = \max_{1 \leq j \leq m} cycletime(P_{alloc(d_j)})$.

Under the one-port model without overlap (*op*), each processor successively receives a data set, performs computations corresponding to the stages assigned to it, and finally outputs the result. Thus its cycle-time is the sum of its communication times and computation time, and the period becomes:

$$\mathcal{P}^{(int-op)} = \max_{1 \leq j \leq m} \left\{ \frac{\delta_{d_j - 1}}{b_{alloc(d_j-1), alloc(d_j)}} + \frac{\sum_{i=d_j}^{e_j} w_i}{s_{alloc(d_j)}} + \frac{\delta_{e_j}}{b_{alloc(d_j), alloc(e_j+1)}} \right\} \quad (2.3)$$

Note that $alloc(d_j - 1) = alloc(e_{j-1}) = alloc(d_{j-1})$ for $j > 1$ and $d_1 - 1 = 0$. Also, $e_j + 1 = d_{j+1}$ for $j < m$, and $e_m + 1 = n + 1$. We still assume that $alloc(0) = 0 = in$ and $alloc(n+1) = p + 1 = out$.

However, if we consider the bounded multi-port model with overlap (*mp*), the processor can simultaneously receive, compute and send, thus the sum in the above formula becomes a maximum. Each processor is only performing one input and one output communication, thus the bandwidth for this communication is the minimum between the link bandwidth and the network card bound.

$$\begin{aligned} \mathcal{P}^{(int-mp)} = \max\nolimits_{1 \leq j \leq m} \Bigg\{ \quad & \max \Bigg(\frac{\delta_{d_j-1}}{\min\Big(b_{alloc(d_j-1), alloc(d_j)}, B^i_{alloc(d_j)}\Big)}, \\ & \frac{\sum_{i=d_j}^{e_j} w_i}{s_{alloc(d_j)}}, \\ & \frac{\delta_{e_j}}{\min\Big(b_{alloc(d_j), alloc(e_j+1)}, B^o_{alloc(d_j)}\Big)} \Bigg) \Bigg\} \end{aligned} \quad (2.4)$$

Note that these formulas are compatible with the formal definition of Equation (2.1). Indeed, in such a case the operation list which minimizes the period or the latency can be defined from a mapping, and these minimum period/latency can be computed through a formula, which renders the corresponding optimization problems tractable.

III.3.2 Replication for reliability

To improve execution reliability, the idea is to replicate stages onto several processors. Such a replication is aimed at increasing the probability of a successful execution. If two processors are assigned the same monolithic stage, both will execute all data sets for this stage[8]. Here we deal with replication oriented towards reliability, where computations are replicated on several processors so as to maximize the probability of getting the results even if some processor(s) would fail. As stated in Section II.3, we associate a failure probability $0 \leq f_u \leq 1$, $1 \leq u \leq p$ to each processor. The failure probability can then be computed given the number m of intervals and the set of processors assigned to each interval (set $alloc(d_j)$ for interval j, $1 \leq j \leq m$):

$$\mathcal{F}^{(int-fp)} = 1 - \prod_{1 \leq j \leq m} \big(1 - \prod_{u \in alloc(d_j)} f_u\big) \tag{2.5}$$

Equation (2.5) is easy to derive: the execution of an interval will succeed except when all its assigned processors fail, and the whole execution will succeed only if all intervals succeed.

When replicating for reliability, we need a standard consensus protocol to determine which of the surviving processors performs the outgoing communications [119]. In other words, such a consensus protocol allows to pay only one incoming communication (whichever comes first) while outgoing communications are serialized (feed all processors assigned to the next interval). Equations (2.2), (2.3) and (2.4) must be revisited because of these extra communications.

For the latency we need to take the longest path, in the worst case scenario of processor failures. Thus, for each interval j, we assume that the surviving processor P_u elected by the consensus is the one that takes the longest time to compute and perform all its outgoing communications. Moreover, we assume that it is the last one to receive the input data: the elected processor for interval $j-1$ serializes its output communications and in the worst case, the last processor to be served is P_u. The following formula returns the more pessimistic latency:

$$\mathcal{L}^{(int-fp)} = \sum_{u \in alloc(1)} \frac{\delta_0}{b_{in,u}} + \sum_{1 \leq j \leq m} \max_{u \in alloc(d_j)} \left\{ \frac{\sum_{i=d_j}^{e_j} w_i}{s_u} + \sum_{v \in alloc(e_j+1)} \frac{\delta_{e_j}}{b_{u,v}} \right\} \tag{2.6}$$

For the period we pay a single incoming communication, and we replace the single outgoing communication by the sum of the communications to all processors responsible for the next interval. The worst case is easy to derive through a formula: for each processor we account for the slowest incoming communication link, and we assume that this processor has been chosen to perform the output communications, and thus account for all outgoing communications. We give the formula only in the

[8]Note that this replication is different from the replication of a non-monolithic stage.

one-port model without overlap; indeed it is quite similar for the bounded multi-port model, except that the sum of input communications, output communications, and computations is replaced by a maximum, and network card bandwidths are added into the formula.

$$\mathcal{P}^{(int-fp)} = \max_{1\leq j\leq m} \max_{u\in alloc(d_j)} \left\{ \frac{\delta_{d_j-1}}{\min\limits_{v\in alloc(d_j-1)} b_{v,u}} + \frac{\sum_{i=d_j}^{e_j} w_i}{s_u} + \sum_{v\in alloc(e_j+1)} \frac{\delta_{e_j}}{b_{u,v}} \right\} \tag{2.7}$$

As already pointed out, these expressions of period and latency are a worst case scenario, and the achieved period and/or latency of the application may well be much better. However, the optimization problem is formally defined as the minimization of the worst case period or latency. Indeed, when defining the operation list for the formal definition of Equation (2.1), we need constraints to be satisfied for all failure scenarios.

III.3.3 Replication for period and latency

In this section, we first consider dealable stages, and then we move to data-parallel stages. In both cases, we discuss the impact of replication on the computation of the operation list which minimizes the period or the latency, for both communication models (bounded multi-port with overlap and one-port without overlap).

Dealable stages. If a stage is dealable, several consecutive computations can be mapped onto distinct processors, and data sets are processed in a round-robin fashion by these processors. The computations of a replicated stage can be fully sequential for a given data set, as long as they do not depend upon previous results for other data sets. We can also replicate an interval of dealable stages. In such a case, different data sets will be processed by different processors, and the latency can be computed as the longest path taken by a data set. Thus, replicating a stage or an interval of stages cannot decrease latency. However, each processor gets fewer data sets to process, thus the period may decrease. On the other hand, the replication of a data-parallel stage can decrease both the latency and the period, at the price of consuming several resources for a given stage.

Defining the cost model for replicated stages is difficult, in particular when two or more consecutive intervals are replicated onto several (distinct) processor sets, and under the one-port model without overlap. Let us start with the replication of a single stage: assume that S_i is replicated onto processors P_{q_1} to P_{q_k}. What is the time needed to process this stage? Because P_{q_1} to P_{q_k} execute the stages in round-robin fashion, the processing time will not be the same for each data set, and we define the traversal time $trav_i$ of S_i as the longest time taken by any processor. With no communication, $trav_i$ is easy to compute from the processor speeds, and the period is then equal to $\frac{trav_i}{k} = \frac{w_i}{k \min_{1\leq u\leq k} s_{q_u}}$, because each processor computes every k-th data set: the slowest processor has indeed $trav_i$ time-steps available between the arrival of two consecutive inputs. However, it is difficult to write formulas for $trav_i$ when there are communication times. If the stages before and after S_i are not replicated, the source of the input and the destination of the output remain the same for each assigned processor P_{q_u} $(1 \leq u \leq k)$, which does simplify the estimation: we would define $trav_i$ as the longest time needed for a processor to receive a message from the source, perform its computations and output the message to the destination. But if, for instance, the stage before S_i is replicated, or belongs to a replicated interval, the source of the input will vary from each processor assigned to the latter stage, and it becomes tricky to analyze the time needed to send and receive messages between any processor pair. We can always take the longest path over all possible pairs, but there may appear synchronization issues that complicate the estimation. Indeed, in some cases, there is no critical resource, and thus we do not know how to estimate the period with an analytical formula. We discuss and exhibit such cases in Chapter 4, where we use a model based on timed Petri nets in order to compute the period of a mapping with replicated dealable stages and communications.

With no constraint on the period, however, the latency can be computed as a longest path: every data set may take a different execution path, and the latency is constrained by the data set which takes the longest time. Resources are not shared between data sets since we can always enforce a period equal to the latency, and thus no more than one data set is processed by the application at any time. Note that if a threshold period must be respected (as in bi-criteria problems), then conflicts of resources used by different data sets are likely to occur, hot spots and synchronization issues appear again, and the story becomes much more complex.

Data-parallel stages. When introducing data-parallel stages, even the computational model requires some attention. Consider a stage S_i to be data-parallelized on processors P_{q_1} to P_{q_k}. We could assume that a fraction of the computations is inherently sequential, hence cannot be parallelized, and thus introduce a fixed overhead o_i that would depend only on the stage and not on the assigned processors. Assuming that each processor executes a share of the work proportional to its speed, the time required to compute S_i on one data set is then $o_i + \frac{w_i}{\sum_{u=1}^{k} s_{q_u}}$. In this case, modeling communication costs is even more difficult than for replicated stages. First, we need to model intra-stage communications. For example we can envision that a given processor, say P_{q_1}, acts as the master and delivers some internal data to the remaining processors P_{q_2} to P_{q_k}, which in turn will return their partial results to P_{q_1}. This scenario would call for a more sophisticated distribution of the work than a simple proportional sharing, because some fast computing processor P_{q_j} may well have a low bandwidth link with P_{q_1}. In addition, inter-stage communications, i.e., input and output data, induce the same difficulties as for replicated stages, as they originate from and exit to various sources. The next difficulty would be to chain two dependent data-parallel stages on two distinct processor sets, which calls for a precise model of redistribution costs.

Model with no communication. Altogether, we see that it is very difficult to come with a satisfactory model for communications, and that replicated and data-parallel stages dramatically complicate the story. For such applications, we present in Chapter 3 complexity results for a very simplified model, where all communication costs and overheads are neglected. This is the price to pay in order to be able to express the period and the latency of a mapping with an analytical formula. We agree that such a model may be realistic only for large-grain applications. In fact, our objective is to assess the inherent difficulty of the period and/or latency optimization problems with replication, and we believe that the complexity results established in this framework will provide a sound theoretical basis for more experimental approaches.

Replication for performance versus replication for reliability. In this section, we have discussed the difficulties induced by replication for performance, i.e., dealable or data-parallel stages, when there are communications. However we point out that it is possible to mix replication for reliability (redundant computation of data sets) and replication of dealable or data-parallel stages (round-robin or parallel computation of data sets), even though it mixes the difficulties of both models. For those cases which are tractable, we detail the corresponding formulas in the following section.

III.3.4 Optimization problems for General mappings

We have seen above that optimization problems for one-to-one and interval mappings can be formalized in most of the cases. The difficulties arise from the introduction of data-parallel stages (both for latency and period definition), and also the definition of the period with replication of dealable stages, in particular when communications are heterogeneous (*Fully Heterogeneous* platforms).

If we move to general mappings, some of the previous problems become even more complicated, because a processor may have several distinct computations and communications to perform, and it may be difficult to decide how to order these communications in the operation list. The failure probability and latency can still be relatively easily defined, while the period causes more problems.

Failure probability. First, we generalize the definition of the failure probability function, including all kinds of replication. Recall that processors are organized into teams, and a subset of stages is allocated to each team. A team either processes the whole subset of stages, or it is processing a round of a deal on this subset, or finally it can handle a fraction of a data-parallel computation (if the subset is reduced to a single stage). In all cases, the application will be successful if and only if there is at least one surviving processor per team. Note that this expression works because we assume that one processor can be involved only in one single team, according to our definition of general mappings. Thus we have:

$$\mathcal{F}^{(gen)} = 1 - \prod_{1 \leq j \leq m} \prod_{1 \leq k \leq t_{d_j}} (1 - \prod_{u \in T_{d_j,k}} f_u) \tag{2.8}$$

Latency. The latency can also be defined for general mappings in a mono-criterion problem: we can assume that a data set enters the pipeline only once the previous one has been output, thus we avoid any conflicts during execution. The latency is then defined as the longest path taken by a data set (recall that paths may differ from one data set to another because of replication).

For $1 \leq i \leq n$, we set $\Delta_i = 1$ if S_{i-1} and S_i are in the same subset, i.e., $\exists j \in [1..m]$ such that $i - 1 \in stages_j$ and $i \in stages_j$. Otherwise, $\Delta_i = 0$. Then, for $1 \leq i \leq n$, we denote by $trav_i$ the maximum traversal time for S_i. If $type_i \neq dp$, i.e., S_i is not data-parallel, then the traversal time is constrained by the slowest processor computing S_i: $trav_i = \frac{w_i}{min_{u \in alloc(i)} s_u}$.

When S_i is data-parallel, the computation is parallelized and the effective speed of the computation is obtained by summing the speeds of the slowest processor in each team. Recall that $T_{i,k}$ denotes the set of processors in the k-th team processing S_i. We have $trav_i = \frac{w_i}{\sum_{1 \leq k \leq t_i} min_{u \in T_{i,k}} s_u}$. However, as discussed previously, it is difficult to account for communications in a data-parallel scheme, and the communication time to the various processors is difficult to estimate. Thus this traversal time is valid only in a case with no communication and no start-up overhead. In such a case, the latency is the sum of the traversal times, $\mathcal{L} = \sum_{1 \leq i \leq n} trav_i$.

We are then ready to write the latency formula which is valid for *Communication Homogeneous* platforms if there is no data-parallelism. Note that communications to stage S_i are paid as many times as the number of processors in the target team. Also, one final communication is paid to output the result.

$$\mathcal{L}^{(gen)} = \sum_{1 \leq i \leq n} \left(\max_{1 \leq k \leq t_i} \left\{ \Delta_i |T_{i,k}| \frac{\delta_{i-1}}{b} + \frac{w_i}{\min_{u \in T_{i,k}} s_u} \right\} \right) + \frac{\delta_{n+1}}{b} \tag{2.9}$$

No formula can be written for *Fully Heterogeneous* platforms because of the complexity added by heterogeneous communications. We could easily replace the constant bandwidths b in Equation (2.9) by the minimum bandwidth between all couples of processors involved in the communication. However, this would be an upper bound of the latency, since slower links might be compensated by faster processors in the execution paths. Moreover, it is easy to compute the exact latency in polynomial time, since it only amounts to computing a longest path in a directed acyclic graph.

A difficulty arises when considering latency in a bi-criteria setting in which we also try to minimize the period. If a fixed period must be respected, then conflicts can occur, the ordering of communications is vital, and problems similar to those encountered with the period must be tackled. We now discuss the period for general mappings.

Period. We already explained why it is difficult to express the period of a linear chain application mapping with replication as an analytical formula, even for the simpler case of INTERVAL mappings. We now focus on general mappings with no replication for period and latency, but only replication for failure. Under the bounded multi-port model with overlap, the period is defined as the maximum cycle-time of a processor, and since all communications can occur in parallel, a processor P_u is performing all its input communications on, say, data sets $k_1+1, \dots, k_\ell+1$, while it is computing data sets $k_1, \dots, k_\ell$ and sending the result for data sets $k_1-1, \dots, k_\ell-1$. Here, ℓ is the number of intervals of consecutive stages that P_u has been assigned, and the processor works simultaneously on various data sets, hence avoiding conflicts. This way, we can guarantee that the optimal period is reached, and come up with a valid operation list satisfying the definition of Equation (2.1). Formally, the period is therefore defined as:

$$
\begin{aligned}
\mathcal{P}^{(gen-mp)} \quad = \quad & \max_{1 \le j \le m} \; \max_{u \in alloc(d_j)} \Bigg\{ \\
\max \quad & \Bigg(\max_{i \in stages_j} \; \max_{v \in alloc(i-1)} \Delta_i \frac{\delta_{i-1}}{b_{v,u}}, \; \sum_{i \in stages_j} \Delta_i \frac{\delta_{i-1}}{B^i_u}, \\
& \frac{\sum_{i \in stages_j} w_i}{s_u}, \\
& \max_{i \in stages_j} \; \max_{v \in alloc(i+1)} \Delta_{i+1} \frac{\delta_i}{b_{u,v}}, \; \sum_{i \in stages_j} \Delta_{i+1} \frac{\delta_i}{B^o_u} \Bigg) \Bigg\}
\end{aligned}
\tag{2.10}
$$

The first line represents constraints on input communications: the period is greater than the maximum incoming communication of subset j, and also the sum of incoming communications is constrained by the input network bandwidth card of the processor. An incoming communication is paid if and only if the previous stage is not in the same subset, i.e., $\Delta_i = 1$. Similarly, the third line expresses constraints on output communications. The second line ensures that the period is greater than the computation time for each subset.

However, in the model without overlap, conflicts arise similarly to the case with replication because of communications, and it is actually NP-hard to decide in which order communications should be performed (i.e., find a valid operation list) in order to obtain the minimum period[9].

[9]This NP-completeness proof is part of recent work with Fanny Dufossé, Loïc Magnan and Yves Robert, and will soon appear as a research report on my webpage. In the meantime, please ask me for details.

IV Summary and conclusion

In this chapter, we have presented a realistic model to express multi-criteria problems for the scheduling of linear chain applications. We introduced replication for reliability, and also replication for performance with dealable and data-parallel stages. We described two communication models, the one-port model without overlap, and the bounded multi-port model with overlap.

In most cases, we were able to formally define the period, latency and failure probability of a given mapping as an analytical formula, and in particular when restricting to a class of simple but realistic mappings, such as INTERVAL mappings. This allowed us to circumvent the formal definition of Equation (2.1), based on the operation list. However, we could not come up with a reasonable model for data-parallel stages when taking communications into account, and it seems difficult to handle dealable stages on *Fully Heterogeneous* platforms (we will exhibit in Chapter 4 cases in which, given an allocation function, the operation list which minimizes the period can be found in polynomial time, and cases in which this problem is NP-hard).

When considering GENERAL mappings, and restricting to replication for reliability (which is much easier to handle than replication for performance), problems became even harder because a processor is no longer involved in a single communication. Under the bounded multi-port model with overlap, the period can still be defined with an analytical formula, while it turns out that finding the minimum period (i.e., a valid operation list which minimizes the period) of a given mapping is NP-hard under the one-port model with no overlap. So guess the difficulty of finding the best mapping! Anyway, we are now ready to tackle this additional difficulty, for cases in which we can come up with a formula to define our three optimization criteria: failure probability, latency and period. This is the goal of Chapter 3.

n	number of stages
S_i	stage i
δ_i	size of data output for stage i
w_i	amount of computations for stage i
p	number of processors
P_u	processor u
P_{in}, P_{out}	special processors for input/output, $P_{in} = P_0$ and $P_{out} = P_{p+1}$
s_u	speed of processor P_u
f_u	failure probability of P_u
B_u^i	input network card capacity of P_u
B_u^o	output network card capacity of P_u
$link_{u,v}$	bidirectional link between P_u and P_v
$b_{u,v}$	bandwidth of link $link_{u,v}$
m	number of intervals (or subsets)
$I_j = [d_j, e_j]$	interval j
$stages_j$	stage indices in subset j; $d_j \in stages_j$
Δ_i	equals 1 if S_{i-1} and S_i are in the same subset, 0 otherwise
$alloc(i)$	processor, or set of processor indices, assigned to S_i
t_i	number of teams for S_i
$T_{i,k}$	k-th team working on S_i $(1 \leq k \leq t_i)$
$type_i$	stage type, equal to *dp* or *deal*, if replicated
o_i	data-parallelization overhead
$trav_i$	traversal time of S_i
$\mathcal{L}$	latency
$\mathcal{P}$	period
$\mathcal{F}$	failure probability

Table 2.1: Notations for Chapters 2 and 3.

CHAPTER 3

Complexity results for linear chain applications

In this chapter, we present complexity results for the various optimization problems defined in Chapter 2. We are interested in the scheduling of linear chain applications, onto computational platforms which range from *Fully Homogeneous* to *Fully Heterogeneous*. We have introduced many optimization criteria, and in particular we have identified cases in which we were able to express the failure probability, the latency, and the period of a given mapping as an analytical formula (recall that finding the operation list which minimizes the period for a given maping is NP-hard in some cases).

We first tackle mono-criterion optimization problems in Section I, for those cases which were identified as tractable in the previous chapter. Then we add more challenges with a study of bi-criteria optimization problems, i.e., we optimize one criterion while a threshold should not be exceeded for the other one (Section II). The chapter ends with a short summary and conclusion in Section III.

Recall that all notations are summarized in Table 2.1 (page 33).

I Mono-criterion problems

In this section, we address mono-criterion problems by increasing order of difficulty. We start with the failure probability in Section I.1, which can be easily solved in a mono-criterion setting, and only raises some issues for One-to-one mappings. Then we address in Section I.2 the problem of latency, which is not difficult for *Communication Homogeneous* platforms with no replication, but surprisingly turns out more tricky on *Fully Heterogeneous* platforms or when considering data-parallel stages. Finally, we consider in Section I.3 the period minimization problem, for which it is in some cases NP-hard to even compute the period of a given mapping, i.e., find a valid operation list which minimizes the period.

I.1 Failure probability

The failure minimization problem turns out simple for Interval and General mappings:

Theorem 1. *Minimizing the failure probability can be done in polynomial time for* Interval *and* General *mappings.*

Table 3.1: Summary of complexity results for failure probability.

	Failure Hom.	***Failure Het.***
One-to-one	polynomial	NP-hard
Interval	polynomial	
General	polynomial	

Proof. This can be seen easily from Equation (2.8) which states that in the most general case,

$$\mathcal{F} = 1 - \prod_{1 \le j \le m} \prod_{1 \le k \le t_{d_j}} (1 - \prod_{u \in T_{d_j,k}} f_u).$$

The minimum is reached by replicating the whole pipeline as a single interval ($m = 1$) consisting in a single team ($t_1 = 1$) on all processors ($T_{1,1} = \{1, \dots, p\}$), thus obtaining $\mathcal{F} = \prod_{u=1}^{p} f_u$. This is true for all stage and platform types. □

Note that for this criterion, there is no need to assign dealable or data-parallel stages onto several distinct processors, since this kind of replication only affects the period and the latency. This result is naturally retrieved from the failure probability formula, which states that no replication for performance should be performed in order to minimize the failure probability.

The problem turns out slightly more complex for One-to-one mappings.

Theorem 2. *Minimizing the failure probability can be done in polynomial time for* One-to-one *mappings if the platform is Failure Homogeneous; the problem is NP-hard for Failure Heterogeneous platforms.*

Proof. For *Failure Homogeneous* platforms, if we restrict to one-to-one mappings, the failure probability formula still indicates that no replication for dealable or data-parallel stages should be used. Then the minimum is reached by balancing the number of processors allocated to each stage, thus for instance allocate processors in round robin to stages.

For *Failure Heterogeneous* platforms, the problem is obviously in NP, and the reduction comes from 3-PARTITION [54]. Given $3n$ positive integers $a_1, ..., a_{3n}$, we build an instance of our problem as a pipeline composed of n stages, and the platform made of $3n$ processors with $f_i = 2^{-a_i}$, for $1 \le i \le 3n$. Since 3-PARTITION is NP-hard in the strong sense, we can encode the a_i in unary, and thus the size of our instance is polynomial in the problem size (the f_i are coded in binary, thus in a size $\log(f_i)$). The $\mathcal{F}$ formula is minimized if failure probabilities are balanced between stages, and a perfect load balancing is achieved when a 3-partition exists (the product of f_i amounts to sum the a_i). □

Table 3.1 summarizes complexity results for the different instances of the failure probability minimization problem. Even though these mono-criterion problems are very simple for the most realistic cases of interval and general mappings, we will see in Section II.2 that bi-criteria problems involving reliability issues are more challenging.

I.2 Latency

In this section we discuss the latency minimization problem. First, note that the replication of dealable stages can only increase latency, so does the replication for reliability. Only the replication of data-parallel stages can impact latency. We first discuss the case with no data-parallelism, and then we introduce it, at the price of suppressing communications (see discussions in Chapter 2).

No data-parallelism. Recall that Equation (2.9) shows how to compute the latency for mappings with no data-parallelism, for *Communication Homogeneous* platforms. Intuitively, latency is small when all communications are zeroed out. This is true for *Fully Homogeneous* and *Communication Homogeneous* platforms.

Theorem 3. *With no data-parallelism, the latency minimization problem is polynomial on Fully Homogeneous and Communication Homogeneous platforms. Finding the* GENERAL *mapping which minimizes the latency is also polynomial on Fully Heterogeneous platforms, while the problem turns out NP-hard for* ONE-TO-ONE *and* INTERVAL *mappings.*

Let us first discuss *Fully Homogeneous* and *Communication Homogeneous* platforms. For interval and general mappings, the optimal solution is to map all stages on the fastest processor: all communications are zeroed out, except input/output ones. For one-to-one mappings, the optimal solution consists in assigning the most computationally expensive stages to the fastest processors, in a greedy manner (largest stage on fastest processor, second largest on second fastest, and so on). The detailed proof can be found in [C31].

This does not work any more for *Fully Heterogeneous* platforms, because of input and output communications. Indeed, it turns out that both one-to-one and interval mapping problems become NP-hard, see [C19] and [J14] for the involved reductions which prove the completeness of these problems. However, finding the general mapping which minimizes the latency on *Fully Heterogeneous* platforms can be done in polynomial time, since it amounts at finding a shortest path in a graph (see [C19]).

With data-parallelism. Since we neglect communication costs, we do consider either homogeneous platforms with same speed processors, or heterogeneous platforms with different speed processors. We have the following result:

Theorem 4. *With data-parallelism and no communication, the latency minimization problem is polynomial on homogeneous platforms. Finding the mapping which minimizes the latency is NP-hard on heterogeneous platforms.*

The proofs, which are technical, can be found in [J10]. In the homogeneous case, we exhibit a complex dynamic programming algorithm for interval mappings.

Table 3.2: Summary of complexity results for latency.

	Fully Hom.	***Comm. Hom.***	***Hetero.***
no DP, One-to-one	polynomial [C31]		NP-hard [C19]
no DP, Interval	polynomial [C31]		NP-hard [J14]
no DP, General	polynomial [C19]		
with DP, no coms	polynomial [J10]	NP-hard [J10]	

This dynamic programming scheme can be rather easily extended to one-to-one and general mappings, thus filling all complexity gaps. For heterogeneous processors, the reduction proposed in [J10] comes from 2-PARTITION, and the instance of our problem consists of a 2-stages pipeline. The corresponding mapping is a one-to-one mapping, which is also an interval and a general mapping, thus the reduction works for all mapping types, which proves the NP-hardness of all problems.

Table 3.2 summarizes complexity results for the different instances of the latency minimization problem.

I.3 Period

We discuss in this section the period minimization problem, for problem instances on which we are able to express the period with a formula. First, note that replication for reliability cannot decrease the period, thus we assume that the optimal mapping does no such replication. However, both replication of dealable stages and data-parallel stages decrease the period, and should be considered.

We start in a scenario with only monolithic stages, i.e., we do not yet consider any replication, and we consider one-to-one mappings.

Theorem 5. *For Fully Homogeneous and Communication Homogeneous platforms, the optimal* One-to-one *mapping with no replication can be determined in polynomial time; the problem becomes NP-hard on Fully Heterogeneous platforms.*

For *Fully Homogeneous* and *Communication Homogeneous* platforms, the optimal mapping can be found with a binary-search algorithm on the period, that iterates until the optimal period is found. At each step of the binary search, a greedy assignment procedure checks whether the period can be achieved or not. No simple greedy algorithm allow to compute the optimal mapping, but the binary search in the above algorithm can be performed in polynomial time. Please refer to [J9] for details of this sophisticated algorithm, which mixes binary search and greedy algorithm, for the one-port model without overlap. Note that a similar algorithm works for the bounded multi-port model with overlap, but that instead of checking whether the sum of communications and computations fits into the period, one may check that the maximum respects the fixed period. Therefore, the result holds true for both communication models.

For *Fully Heterogeneous* platforms, the problem turns out to be NP-hard, see [J9] (the involved reduction comes from the MINIMUM METRIC BOTTLENECK WANDERING SALESPERSON PROBLEM, [6, 44]).

We now proceed to interval mappings, still with a pipeline consisting only of monolithic stages (and thus, no replication). In a simplified case with no communication costs, this scheduling problem is similar to the well-known chains-to-chains problem, which consists in partitioning an array of n elements $a_1, a_2, \dots, a_n$ into p intervals whose element sums are well balanced (technically, the aim is to minimize the largest sum of the elements of any interval). Several algorithms and heuristics have been proposed to solve this load-balancing problem, including [25, 70, 59, 71, 98]. We refer the reader to the survey paper by Pinar and Aykanat [102] for a detailed overview and comparison of the literature. It amounts to load-balance n computations whose ordering must be preserved (hence the restriction to intervals) onto p identical processors. For *Fully Homogeneous* platforms, the period minimization problem remains polynomial even in presence of communications. However, the advent of heterogeneous clusters naturally leads to the following generalization of the chains-to-chains problem: can we partition the n elements into p intervals whose element sums match p prescribed values (the processor speeds) as closely as possible? We established the NP-hardness of this important extension of the chains-to-chains problem, and thus the NP-completeness of the period minimization problem on *Communication Homogeneous* platforms. The involved reduction (see [J9]) comes from the NUMERICAL MATCHING WITH TARGET SUMS problem, which is NP-complete in the strong sense [54].

The optimal interval mapping on *Fully Homogeneous* platforms can be obtained in polynomial time with a dynamic programming algorithm: we recursively compute the optimal period that can be achieved by any interval mapping of stages S_i to S_j using exactly k processors, for $1 \leq i \leq j \leq n$ and $1 \leq k \leq p$. Please refer to [J9] for details[1].

It is not possible to extend the previous dynamic programming algorithm to deal with *Communication Homogeneous* platforms. This is because the algorithm intrinsically relies on identical processors in the recursion. Heterogeneous processors would execute sub-intervals with different cycle-times. Because of this additional difficulty, the INTERVAL problem for *Communication Homogeneous* platforms is NP-hard, see [J9]. Note that the proof considers an application with no communication costs, thus it remains valid whatever the communication model.

Theorem 6. *For Fully Homogeneous platforms, the optimal* INTERVAL *mapping with no replication can be determined in polynomial time; the problem is NP-hard on Communication Homogeneous and Fully Heterogeneous platforms.*

For general mappings, we restrict to the bounded multi-port model with overlap (otherwise we do not have a formula to express the period of a given mapping). In

[1]Here again, the proof was originally written for the one-port model without overlap, but the extension to the bounded multi-port model with overlap is not difficult.

Table 3.3: Summary of complexity results for period.

	Fully Hom.	***Comm. Hom.***	***Hetero.***
ONE-TO-ONE	polynomial [J9]	poly., NP-hard (rep) [J9]	NP-hard [J9]
INTERVAL	polynomial [J9]	NP-hard [J9]	NP-hard [J9]
GENERAL	NP-hard, poly. (rep)	NP-hard	

this case, even for a simple application with no communication costs, the problem is NP-hard for *Fully Homogeneous* platforms. Indeed, consider a pipeline consisting in n monolithic stages with computation costs a_i, and two identical processors. It is straightforward to see that the general mapping problem is equivalent to 2-PARTITION [54], which leads to the following theorem:

Theorem 7. *Finding the optimal* GENERAL *mapping with no replication under the bounded multi-port model with overlap is NP-hard on all platform types.*

Table 3.3 summarizes complexity results for the different instances of the period minimization problem. General mapping problems are all NP-hard, for those instances for which the period can be expressed as a formula. We see that one level of heterogeneity (in processor speed) is enough to make interval mapping problems NP-hard, while two levels of heterogeneity (adding different link bandwidths) are required to make one-to-one mapping problems NP-hard as well.

When adding replication, almost all problems remain of the same complexity. However, the polynomial case of one-to-one mappings on *Communication Homogeneous* platforms becomes NP-hard, and the NP-hard case of general mappings on *Fully Homogeneous* platforms becomes polynomial.

The new NP-completeness proof can be found in [J10]. The reduction comes from 2-PARTITION [54], and we build a 2-stages pipeline for which we enforce that data-parallelism should used (because of different-speed processors, replication would waste resources, since all processors synchronize on the slowest one).

Then, all other NP-completeness proofs on *Communication Homogeneous* platforms can be adapted. For one-to-one and interval mappings, we can enforce that only one processor can be used per interval, and thus no replication is possible. For general mappings, we can enforce that data-parallelism must be used, by having differences in processor speeds, such that the waste of resources in case of replication of dealable stages is not acceptable. This does not work for general mappings on *Fully Homogeneous* platforms, but this case becomes easy, similarly to the interval mapping case: with replication, the interval or general mapping which minimizes the period consists in replicating the whole pipeline as a single interval on all processors, see [J10]. The problem is thus of polynomial complexity.

Finally, for one-to-one mappings on *Fully Homogeneous* platforms, we assign at first one processor per stage, and then we add processors to the stages with the greatest remaining period, until all processors are distributed. It is easy to see that this greedy algorithm returns the optimal period. This fills all complexity gaps.

II Bi-criteria problems

Now that the complexity of all mono-criterion problems has been established, we are ready to deal with bi-criteria objective functions. As stated in Chapter 2, we aim at minimizing one objective under threshold constraints on the other objective, which seems more natural than the minimization of a linear combination of both criteria. We focus in this section to the case of interval mappings, which are both realistic and tractable. Also, we do not investigate trade-offs between replication for failure and replication for performance, and we consider that all stages are monolithic. Extensions will be discussed in Section III.

In Section II.1, we aim at minimizing the latency under period constraints (or the converse). Because most problem instances are NP-hard, we provide a formulation in terms of an integer linear program.

In Section II.2, we investigate trade-offs between latency and failure probability. We succeed in identifying problem instances that have polynomial complexity. For those problems which are NP-hard, we cannot even cast the problem in terms of a linear program, because of the strong non-linearity of Equation (2.8).

We do not deal with the last combination, i.e., period/reliability optimization problems, because both difficulties sum up: period minimization is NP-hard in most instances, hence the bi-criteria problem is NP-hard as well, and an integer linear program formulation seems unreachable.

II.1 Period and latency

In this section we deal with the problem of minimizing the period under latency constraints, or the converse. Recall from Section I.3 that the period minimization problem is already NP-hard for interval mappings on *Communication Homogeneous* platforms. Of course, the bi-criteria period/latency interval mapping optimization remains NP-hard for *Communication Homogeneous* and *Fully Heterogeneous* platforms, since the period minimization problem already is NP-hard for such mappings. Note that the bi-criteria problem is polynomial for *Fully Homogeneous* platforms, see the dynamic programming algorithm in [116].

In the following, we present an integer linear program to compute the optimal interval mapping on *Fully Heterogeneous* platforms, respecting either a fixed latency or a fixed period. Recall that we have n stages and p processors, plus two fictitious extra stages S_0 and S_{n+1} respectively assigned to two extra processor P_0 and P_{p+1}. First we need to define a few variables:

- For $i \in [0..n+1]$ and $u \in [0..p+1]$, $x_{i,u}$ is a boolean variable equal to 1 if stage S_i is assigned to processor P_u; we let $x_{0,0} = x_{n+1,p+1} = 1$, and $x_{i,0} = x_{i,p+1} = 0$ for $1 \leq i \leq n$.
- For $i \in [0..n]$ and $u \in [0..p+1]$, $y_{i,u}$ is a boolean variable equal to 1 if stages S_i and S_{i+1} are both assigned to P_u; we let $y_{i,0} = y_{i,p+1} = 0$ for all i, and $y_{0,u} = y_{n,u} = 0$ for all u.

- For $i \in [0..n]$, $u, v \in [0..p+1]$ with $u \neq v$, $z_{i,u,v}$ is a boolean variable equal to 1 if stage S_i is assigned to P_u and stage S_{i+1} is assigned to P_v: hence $link_{u,v} : P_u \to P_v$ is used for the communication between these two stages. If $i \neq 0$ then $z_{i,0,v} = 0$ for all $v \neq 0$ and if $i \neq n$ then $z_{i,u,p+1} = 0$ for all $u \neq p+1$.
- For $u \in [1..p]$, $first_u$ is an integer variable which denotes the first stage assigned to P_u; similarly, $last_u$ denotes the last stage assigned to P_u. Thus P_u is assigned the interval $[first_u, last_u]$. Of course $1 \leq first_u \leq last_u \leq n$.
- $\mathcal{O}$ is the variable to optimize, so depending on the objective function it corresponds either to the period or to the latency.

We list below the constraints that need to be enforced. For simplicity, we write $\sum_u$ instead of $\sum_{u=0}^{p+1}$ when summing over all processors.

- Every stage is assigned exactly one processor: $\forall i \in [0..n+1]$, $\sum_u x_{i,u} = 1$.
- For all $i \in [0..n]$, the communication from S_i to S_{i+1} either is assigned a communication link, or both stages are assigned to the same processor:

$$\forall i \in [0..n], \qquad \sum_{u \neq v} z_{i,u,v} + \sum_u y_{i,u} = 1$$

- If stage S_i is assigned to P_u and stage S_{i+1} to P_v, then $link_{u,v} : P_u \to P_v$ is used for this communication:

$$\forall i \in [0..n], \forall u, v \in [0..p+1], u \neq v, \qquad x_{i,u} + x_{i+1,v} \leq 1 + z_{i,u,v}$$

- If both stages S_i and S_{i+1} are assigned to P_u, then $y_{i,u} = 1$:

$$\forall i \in [0..n], \forall u \in [0..p+1], \qquad x_{i,u} + x_{i+1,u} \leq 1 + y_{i,u}$$

- If stage S_i is assigned to P_u, then necessarily $first_u \leq i \leq last_u$. We write this constraint as:

$$\forall i \in [1..n], \forall u \in [1..p], \qquad first_u \leq i.x_{i,u} + n.(1 - x_{i,u})$$

$$\forall i \in [1..n], \forall u \in [1..p], \qquad last_u \geq i.x_{i,u}$$

- If stage S_i is assigned to P_u and stage S_{i+1} is assigned to $P_v \neq P_u$ (i.e., $z_{i,u,v} = 1$) then necessarily $last_u \leq i$ and $first_v \geq i + 1$ since we consider intervals. We write this constraint as:

$$\forall i \in [1..n-1], \forall u, v \in [1..p], u \neq v, \qquad last_u \leq i.z_{i,u,v} + n.(1 - z_{i,u,v})$$

$$\forall i \in [1..n-1], \forall u, v \in [1..p], u \neq v, \qquad first_v \geq (i+1).z_{i,u,v}$$

The latency of the schedule is bounded by $\mathcal{L}$:

$$\sum_{u=1}^{p}\sum_{i=1}^{n}\left(\left(\sum_{t\neq u}\frac{\delta_{i-1}}{b_{t,u}}z_{i-1,t,u}\right)+\frac{w_i}{s_u}x_{i,u}\right)+\left(\sum_{u=0}^{p}\frac{\delta_n}{b_{u,p+1}}z_{n,u,p+1}\right)\leq\mathcal{L}.$$

There remains to express the period of each processor and to constrain it by $\mathcal{P}$:

$$\forall u\in[1..p],\quad \sum_{i=1}^{n}\left(\left(\sum_{t\neq u}\frac{\delta_{i-1}}{b_{t,u}}z_{i-1,t,u}\right)+\frac{w_i}{s_u}x_{i,u}+\left(\sum_{v\neq u}\frac{\delta_i}{b_{u,v}}z_{i,u,v}\right)\right)\leq\mathcal{P}.$$

Finally, the objective function is either to minimize the period $\mathcal{P}$ respecting the fixed latency $\mathcal{L}$ or to minimize the latency $\mathcal{L}$ with a fixed period $\mathcal{P}$. So in the first case we fix $\mathcal{L}$ and set $\mathcal{O}=\mathcal{P}$. In the second case $\mathcal{P}$ is fixed a priori and $\mathcal{O}=\mathcal{L}$. With this mechanism the objective function reduces to minimizing $\mathcal{O}$ in both cases.

We have $O(np^2)$ variables, and as many constraints. All variables are boolean or integer, except the period and latency, which are rational. Heuristics and experiments for the period/latency problem can be found in [C18],[C21].

II.2 Latency and reliability

In this section we deal with the problem of minimizing the latency under reliability constraints, or the converse. As in the previous section, we restrict to interval mappings which correspond to the more realistic cases.

We start with a preliminary lemma which proves that there exists an optimal solution of both bi-criteria problems consisting of a single interval for *Fully Homogeneous* platforms, and for *Communication Homogeneous-Failure Homogeneous* platforms.

Lemma 1. *On Fully Homogeneous and Communication Homogeneous-Failure Homogeneous platforms, there is a mapping of the pipeline as a single interval which minimizes the failure probability under a fixed latency threshold, and there is a mapping of the pipeline as a single interval which minimizes the latency under a fixed failure probability threshold.*

The proof can be found in [C19]. We point out that this lemma cannot be extended to the case *Communication Homogeneous* and *Failure Heterogeneous*. Figure 3.1 provides a counter-example application; the latency threshold is fixed to 22, and the target platform consists of 11 processors with the following characteristics:

- processor P_1 is slow but reliable: $s_1=1$ and $f_1=0.1$;
- the 10 remaining processors P_u, $2\leq u\leq 11$, are fast but unreliable: $s_u=100$ and $f_u=0.8$;
- all communication links have a bandwidth $b=1$.

Let us consider a mapping consisting of a single interval. Then the slow processor P_1 cannot be used in the replication scheme, otherwise the latency is greater than

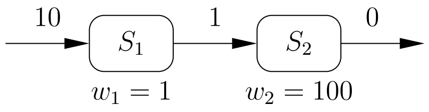


Figure 3.1: Counter-example with 2 intervals.

$10+101/1 > 22$. Also, if we use three fast processors, the latency is $3*10+101/100 > 22$. Thus the best solution uses two fast processors and has a failure probability of $1-(1-0.8^2) = 0.64$, which is very high. We can do much better with two intervals, by using the slow processor on the slow stage, and then replicate ten times the second stage on the fast processors, achieving a latency of $10 + 1/1 + 10 * 1 + 100/100 = 22$ and a failure probability of $1-(1-0.1).(1-0.8^{10}) < 0.2$. Thus the optimal solution does not consist of a single interval in this case.

For *Fully Homogeneous* platforms, we consider that all failure probabilities are identical, since the platform is made of identical processors. We have seen from Lemma 1 that the optimal solution for a bi-criteria mapping on such platforms always consists in mapping the whole pipeline as a single interval. Otherwise, both latency and failure probability would be increased. It is then easy to derive a polynomial time algorithms which solves the bi-criteria problems. Informally, the algorithms find the maximum number of processors k that can be used in the replication set, and the whole interval is mapped on a set of k identical processors. With different failure probabilities, the more reliable processors would be used.

For *Communication Homogeneous* platforms, we first consider the simpler case where all failure probabilities are identical (*Failure Homogeneous*). In this case, the optimal bi-criteria solution still consists of mapping the pipeline as a single interval (see Lemma 1), and polynomial time algorithms can solve the problem. Informally, we add the fastest processors to the replication set while the latency is not exceeded (or until $\mathcal{F}$ is reached), thus reducing the failure probability and increasing the latency.

However, the problem is more complex when we consider different failure probabilities (*Failure Heterogeneous*). It is also more natural since we have different processors and there is no reason why they would have the same failure probability. Unfortunately, we can exhibit in this case problem instances for which an optimal solution necessarily consists of several intervals (see the example of Figure 3.1). Therefore, the previous algorithms cannot work anymore. The complexity of the problem remains open, but we conjecture it is NP-hard, since there is a clear trade-off between the use of reliable processors versus the use of fast processors.

For *Fully Heterogeneous* platforms, since the latency minimization problem is known to be NP-hard, both bi-criteria problems also are NP-hard.

III Summary and conclusion

In this chapter, we have been able to assess the complexity of several instances of the linear chain application mapping problem. Through this exhaustive study, we have established the inherent difficulties raised by each criterion, be it the failure probability, the latency, or the period. Replication was rendering the modeling process difficult in Chapter 2, and in this chapter we were able to exhibit cases in which the addition of replication transforms a problem that can be solved in polynomial time into a problem that is NP-hard (see, for *Communication Homogeneous* platforms, the latency minimization problem, or one-to-one mappings to minimize the period).

We have not done an exhaustive study of multi-criteria mapping problems, but most of them are already NP-hard (as soon as the optimization problem for one of the criteria is NP-hard). The two case-studies that we selected allowed us first to show an example of multi-criteria integer linear program, aiming at solving NP-hard problems, and second to illustrate the additional complexity of mixing two criteria which are, alone, polynomial, if not trivial.

Of course, many more combinations can be studied, see for instance [J10] for bi-criteria problems period/latency with replication and no communication. Some of them are still open, for instance those involving one-to-one mappings. Also, we have not tackled tri-criteria problems in this study, which would mix difficulties of all criteria. We rather envision an experimental approach, through the design of sophisticated heuristics which build upon the knowledge that we gained on each criterion, in order to tackle these complex multi-criteria problems.

In the next chapter, we come back to the problem of computing the minimum period for a given mapping, which turned out to be a difficult problem in Chapter 2, and we propose to address the optimization problems in the more complex setting of dynamic platforms, thus introducing performance models.

CHAPTER 4

Dealing with dynamic platforms

We have seen in Chapter 3 that many mapping problems are NP-hard, and it is sometimes even difficult to compute the period achieved by a given mapping (see Chapter 2). We investigate in this chapter the use of stochastic performance models in order to evaluate the period of a mapping. Moreover, these models aim at extending the previous study to dynamic platforms.

In a brief introduction (Section I), we motivate the need to tackling dynamic platforms. Then we present two performance models, the first one based on Performance Evaluation Process Algebra (Section II), and the second one using timed Petri nets (Section III). The main characteristics of each model are detailed, as well as their utility and the ultimate goal of this modeling. An important feature of dynamic platforms is that they might be subject to failures, and a particular care must be brought on this point. We have discussed in Chapters 2 and 3 a failure model which was fitting the pipelined application scenario, independent of time. We present in Section IV a different failure model, targeting the scheduling of a divisible workload on a dynamic platform subject to unrecoverable interruptions. Finally, we conclude this chapter in Section V.

I Static versus dynamic platforms

Static platforms are characterized by their limited heterogeneity (processor speed, bandwidth of the communication links, etc.). Moreover, the characteristics of the platform are supposed to be static (or to change at very slow rate) over time. A typical example of such platforms is a computational grid dedicated to one user, or to a small group of well identified users, on which are running a small number of different applications, also well identified. Another example is a set of resources provided by a team of users, and shared by these users. These resources are assumed to be temporary dedicated to a specific application, or to a well identified small set of applications. Static platforms engender several difficulties due to their heterogeneity, and their hierarchical nature. Recently, several models [29, 15, 65, 17] have been proposed in the literature for modeling the topology and the resources of such platforms. The important issues concern their ability to describe heterogeneous resources (both processing and communication resources) and the interferences between simultaneous activities. In particular, the following questions must be answered: is it possible for a processor to be involved in more than one incoming (and/or) one outgoing communication? is it important to take into account the interferences that can occur between several distinct point to point communications? Are processing performances affected by simultaneous incoming (and/or) outgoing communications? Note that these questions are important but are limited to assessing the static performance of platform parameters. While analyzing different communication models in Chapter 2, we have given elements of answer. Thus, in the previous chapters, we have focused on such platforms, which were assumed to have a static behavior but already raised many challenges.

On the other hand, *dynamic platforms* are characterized by their larger size, and greater heterogeneity. Still, the resources of these platforms (topology, message routes, etc.) and their characteristics are assumed to be known by a centralized control mechanism, even though they change over time. Radical changes are caused by failures, but the performance of a processor can also be slightly reduced because, for instance, another user launched a new process onto this resource. A typical example of such platform is a general purpose computational grid, or a set of resources provided by a team of users that can change significantly over time. In this chapter, we point out that dynamic platforms exhibit a great variability of their resources. For instance, processor speeds are expected to evolve over time, as the execution proceeds, and this evolution is likely to take the form of abrupt changes followed by epochs of relative stability. Similarly, link bandwidths, or even availability, is subject to dramatically change during execution. Accounting for the dynamic nature of platform parameters is a key component for the design of robust scheduling algorithms.

If we want to model variability, we need to express the fact that exact data about processor speeds and network links cannot be obtained from a given platform. The measured run-time of a function may vary, depending on the load of the processor on which it is executed and many other factors. Also, the processor may be subject to failures, and so the measured run-time may be completely different from the expected run-time. For these reasons, the appropriate modeling formalism seems to be one which uses probabilistic random variables to estimate times. Such a model gives rise to a stochastic process which is analyzed in order to get information about the performance of the system. When we do only know the average duration of run-times, then the appropriate random number distribution to use is the exponential distribution. An important class of stochastic processes are Continuous-Time Markov Chains (CTMCs) [79], which have a finite state space, and whose transition rates between states are exponentially distributed.

In Section II, we investigate the use of stochastic process algebras [62], a high-level modeling language which allows to express CTMCs in a structured and compositional way. A model of a given mapping is proposed, which allows us to predict the throughput of the application [J3],[J4], assuming exponential laws for processor speeds and link bandwidths. With this technique, several mappings can be compared, but it turns out that the predicted throughput can be far from the actual throughput.

Therefore, we investigate the use of timed Petri nets [11] in Section III, which allow us to compute the achieved throughput in a deterministic case. Moreover, thanks to this model, we are able to compute the throughput in cases for which we do not know how to compute it with an analytical formula. This model still needs to be extended in order to account for variability and failure of the application.

Another challenge raised by dynamic platforms is the definition of a pertinent failure model. We already tackled failures for linear chain pipelined applications in the previous chapters, and we present a probabilistic approach to failures in Section IV.

II Performance Evaluation Process Algebra

In this section, we are interested in the mapping of a pipelined linear chain application with deal replication. We have a set of heterogeneous processors, interconnected by a heterogeneous network. We focus on the throughput maximization problem, and hence the performance model aims at computing the throughput (or the period) of a given mapping. The model can represent general mappings, in which a processor can even be involved in a round of several deals, thus extending our definition of general mapping to completely general ones.

Continuous-Time Markov Chains (CTMCs) are widely used to model parallel systems because there are many efficient solution methods for finding their stationary probability distribution, thus easily extracting performance information. They represent a practically useful formalism for modeling and analysis. The stationary distribution is a vector of probabilities expressing the likelihood of the system being in each of the states of its finite state space in the long run. Then, the throughput of the system may be extracted from this stationary distribution. Usually, CTMCs are not used directly as a formal modeling language. It is more convenient to use a higher-level modeling language, and to generate the underlying CTMC from this language. For instance, we can use stochastic Petri nets [95, 37], stochastic automata networks [104], or stochastic process algebras [62], which enforce a structured and compositional vision of the system.

In Section II.1, we exhibit a PEPA (Performance Evaluation Process Algebra) model to describe the mapping of a linear chain application with deal replication. Then we explain in Section II.2 how these models are used through a simple tool to decide which mapping would lead to the best application performance. Finally, we conclude and show the limits of such a model in Section II.3.

II.1 PEPA model for linear chain applications

In this section we present our approach to model the mapping of a linear chain application with deal replication on a heterogeneous platform. The model is expressed in Performance Evaluation Process Algebra (PEPA) [62]. We first briefly introduce PEPA, and then we present a generic model.

II.1.1 Introduction to PEPA

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behavior of components, via the activities they undertake and the interactions between them. Timing information is associated with each activity. Thus, when enabled, an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution which has parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. The component combinators used in the model in Section II.1.2, together with their names and interpretations, are presented informally below.

Prefix: The basic mechanism for describing the behavior of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S.

Choice: The choice combinator captures the possibility of competition between different activities. The component $P + Q$ represents a system which may behave either as P or as Q: the activities of both are enabled. The first activity to complete distinguishes one of them: the other is discarded. The system will behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behavior associated with components. Constants are components whose meaning is given by a defining equation.

Cooperation: In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. The set which is used as the subscript to the co-operation symbol, the *cooperation set* L, determines those activities on which the *co-operands* are forced to synchronize. For action types not in L, the components proceed independently and concurrently with their enabled activities. However, an activity whose action type is in the cooperation set cannot proceed until both components enables an activity of that type. The two components then proceed together to complete the *shared activity.* The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity (for details see [62]). We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified (denoted $\top$) and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronized in the final model.

The dynamic behavior of a PEPA model is represented by the evolution of its components, as governed by the operational semantics of PEPA terms (see [62]). Thus, as in classical process algebra, the semantics of each term is given via a labelled *multi-transition* system (the multiplicities of arcs are significant). In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* and these form the nodes of the *derivation graph* which is formed by applying the semantic rules exhaustively.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is the sum of the activity rates labelling arcs connecting node P to node Q.

It is important to note that in such models the estimated durations of tasks etc. are represented as random variables, not constant values. These random variables are exponentially distributed. Repeated samples from the distribution will follow the distribution and conform to the mean but individual samples may potentially take any positive value. It thus accounts for the variability of the platform.

II.1.2 Mapping model

To model the mapping of a linear chain application, we decompose the problem into three parts: stages, processors and network.

The stages. The first part of the model is the *application model*, which is independent of the resources on which the application will be computed. The application consists of n stages, which are each modelled by a PEPA component $Stage_i$ $(1 \leq i \leq n)$.

When $Stage_i$ **is not replicated**, it executes sequentially (following the one-port model without overlap). As its first activity, it obtains data (activity $move_i$), then processes it (activity $process_{i,1}$[1]), and finally moves the processed data to the next stage (activity $move_{i+1}$):

$$Stage_i \overset{def}{=} (move_i, \top).(process_{i,1}, \top).(move_{i+1}, \top).Stage_i$$

Note that all the rates are unspecified, denoted by the distinguished symbol $\top$, since the processing and move times depend on the resources on which the application is running. These rates will be defined later, in another part of the model.

When $Stage_i$ **is replicated**, let np_i be the number of processors which are part of the deal. Recall that, as motivated in Chapter 2, we enforce cyclic allocation of inputs to processors. For this, we introduce, for each stage which is replicated, a *Source* component and a *Sink* component which interface between the processors and the *move* actions which link this stage to its pipeline neighbours. We denote by *worker* an instance of the stage mapped onto a processor (one round of the deal). Each worker $(1 \leq u \leq np_i)$ first gets an input from the source with an $input_{i,u}$ action, processes it $(process_{i,u})$ then transfers its output to the sink $(output_{i,u})$.

[1]The 1 in the index denotes the first (and only) processor for this stage.

We obtain the definitions $Source_i$, $Sink_i$ and $Worker_{i,u}$, for $1 \leq u \leq np_i$:

$$\begin{aligned}
Source_i &\overset{def}{=} (move_i, \top).(input_{i,1}, \top).\\
&\quad (move_i, \top).(input_{i,2}, \top)\dots.\\
&\quad (move_i, \top).(input_{i,np_i}, \top).Source_i\\
Worker_{i,u} &\overset{def}{=} (input_{i,u}, \top).(process_{i,u}, \top).(output_{i,u}, \top).Worker_{i,u}\\
Sink_i &\overset{def}{=} (output_{i,1}, \top).(move_{i+1}, \top).\\
&\quad (output_{i,2}, \top).(move_{i+1}, \top)\dots.\\
&\quad (output_{i,np_i}, \top).(move_{i+1}, \top).Sink_i
\end{aligned}$$

All the workers are independent, and they are synchronized with the source and the sink via the *input* and *output* actions. We define $LI_i = \{input_{i,u}\}_{1 \leq u \leq np_i}$ and $LO_i = \{output_{i,u}\}_{1 \leq u \leq np_i}$ and thus we obtain:

$$Stage_i \overset{def}{=} Source_i \underset{LI_i}{\bowtie} (Worker_{i,1} \mid\mid \dots \mid\mid Worker_{i,np_i}) \underset{LO_i}{\bowtie} Sink_i$$

Once all the stages have been defined, the linear chain application is then a cooperation of the different stages over the $move_i$ activities, for $2 \leq i \leq n$. The activities $move_1$ and $move_{n+1}$ represent respectively, the arrival of an input into the application, and the transfer of the final output out of the pipeline. They do not represent any data transfer between stages, so they are not synchronized within the application. As above, the rates on the input and output actions are left unspecified. These will be defined elsewhere in the model. Finally, we have:

$$Pipeline \overset{def}{=} Stage_1 \underset{\{move_2\}}{\bowtie} Stage_2 \underset{\{move_3\}}{\bowtie} \cdots \underset{\{move_n\}}{\bowtie} Stage_n$$

The processors. We consider that the application must be mapped to a set of p processors. Each worker is implemented by a given (unique) processor, but a processor may host several workers. In order to keep the model simple, we decide to put information about the processor (such as the load of the processor or the number of stages being processed) directly in the rate $\mu_{i,u}$ of the activities $process_{i,u}$, for $1 \leq i \leq n$ and $1 \leq u \leq np_i$ (these activities have been defined for the components $Stage_i$). Each processor is then represented by a PEPA component which has a cyclic behavior, consisting of sequentially processing inputs for a worker. Some examples follow.

- In a case with no replication and with $n = p$, we map one stage (i.e., one worker) per processor (one-to-one mapping): for $1 \leq i \leq p$,

$$Processor_i \overset{def}{=} (process_{i,1}, \mu_{i,1}).Processor_i$$

- If several workers are hosted by the same processor, we use a choice composition. In the following example ($n = p = 2$, and the first stage is replicated

2 times), the first processor processes the first worker of both stages, and the second processor processes only the second worker of stage 1 (so that stage 1 is distributed across two processors).

$$\begin{array}{rcl} Processor_1 & \overset{def}{=} & (process_{1,1}, \mu_{1,1}).Processor_1 \\ & + & (process_{2,1}, \mu_{2,1}).Processor_1 \\ Processor_2 & \overset{def}{=} & (process_{1,2}, \mu_{1,2}).Processor_2 \end{array}$$

More generally, since all processors are independent, the set of processors is defined as a parallel composition of the processor components:

$$Processors \overset{def}{=} Processor_1 \ || \ Processor_2 \ || \ \ldots \ || \ Processor_p$$

The network. Rather than directly representing the physical structure of the underlying network architecture, our network model is designed to allow us to derive the rates of the logical communication actions (*move*, *input*, *output*) from the Network Weather Service (NWS, [128]) monitored physical processor to processor latency information. Using λ_i for the rate of a $move_i$ and $\lambda I_{i,u}$ and $\lambda O_{i,u}$ for the respective rates of $input_{i,u}$ and $output_{i,u}$ activities, the definition of the network is straightforward.

For example, assuming that only stage i is replicated, we obtain the following definition[2]:

$$\begin{array}{rclcl} Network & \overset{def}{=} & (move_1, \lambda_1).Network & + \ldots + & (move_{n+1}, \lambda_{n+1}).Network \\ & + & (input_{i,1}, \lambda I_{i,1}).Network & + \ldots + & (input_{i,np_i}, \lambda I_{i,np_i}).Network \\ & + & (output_{i,1}, \lambda O_{i,1}).Network & + \ldots + & (output_{i,np_i}, \lambda O_{i,np_i}).Network \end{array}$$

Overall model. Once we have defined the different components of the model, we just have to map the stages onto the processors and the network by using the cooperation combinator. Two cooperation sets are used. L_{net} synchronizes the application and the network (it is the set of all the *move*, *input* and *output* activities), while L_{proc} synchronizes the application and the processors (it is the set of all the *process* activities):

$$Mapping \overset{def}{=} Network \underset{L_{net}}{\bowtie} Pipeline \underset{L_{proc}}{\bowtie} Processors$$

II.2 Using the performance model

The performance models can be solved in order to obtain the throughput of a given mapping. In its current form ([J3],[J4]), the solving tool is a generic analysis component. Its ultimate role would be as an integrated component of a run-time scheduler and re-scheduler, adapting the mapping from application to resources in response to

[2]If some other stages are also replicated, we need to add their *input* and *output* activities into the choice.

changes in resource availability and performance. On initialization, information is obtained from the resources and the network with the help of the Network Weather Service NWS [128]. Some additional information must be provided to the tool via some *description files*. First, we need to provide the names of the processors which are to be used for the application (list of IP addresses). Note that NWS must run on each of these nodes, and secure shell access must be allowed to gather some information about the processors. The first processor on the list is called the *reference processor*. Another file must specify the number of stages n of the pipeline, and the average time required to compute one output for each stage on the reference processor, and finally the size of the data transferred to and from each stage. Finally we define a set of candidate *mappings* of stages to processors. Each mapping specifies where the initial data is located, where the output data must be left and (as a tuple) the processors where each stage is processed. For instance, the tuple $(1, \ (1, 2, 3))$ means that the first stage is on processor 1, and that stage 2 is replicated on processors 1, 2 and 3 respectively. The mapping definition is a set of mappings that we aim to compare to each other.

In order to instantiate the parameters of the model, we then use the Network Weather Service (NWS), which is a distributed system that periodically monitors and dynamically forecasts the performance that various network and computational resources can deliver over a given time interval. The service operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers readings of the instantaneous conditions. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame [128].

The computing power of each processor is evaluated through its CPU frequency, and thus we can compare it to the frequency of the reference processor. This approach is somewhat naive since the performance of the processors depends on many other factors including memory access speed and cache policy. Moreover, it depends on the characteristics of the application. However, using the frequency of the processor gives us a rough idea of its global performance.

One model is then generated from each mapping of the description file. Each model is as described in Section II.1.2. Rates are generated from the information gathered before, and the model generation itself is then straightforward. To compute the μ rates, we need to know how many workers are hosted on each processor, and we then assume that the work sharing between them is equitable. The rates of communication (the variously subscripted λ, λI and λO terms) are derived from the processor to processor latency values (la) obtained from NWS, and from the data size values of the data transferred from one stage to another. However, some of the *move*, *input* and *output* activities may express a *data transfer* from processor u to this same processor ($1 \leq u \leq p$). The latencies $la_{u,u}$ should be set to zero. To ensure the logical behavior of the model, these activities are needed and they cannot be immediate (all activities are timed in PEPA), so we set all these latencies to an arbitrarily small value (10^{-5} milliseconds). Details on the parameter generation can be found in [J3],[J4].

Numerical results can been computed from such models with the Java Version of the PEPA Workbench [56, 58]. The throughput of a mapping can be obtained via a single command line (see [J3],[J4]). When a large number of models must be solved, we use task farming to solve them in a parallel scheme. The most lightly loaded processors are selected with the help of the information gathered previously with NWS, and the jobs are distributed on these processors. The gain is almost equal to the number of processors used, even if a small overhead can be observed due to the time required to dispatch the models and to collect the results. During the resolution, all the results are saved in a single file, and finally we find out which mapping produces the best throughput. This mapping is the one we should use to run the application.

The use of the tool takes usually less than one minute for complex applications running on several processors, even when we consider several models which can be relatively large. The distributed resolution of the models allows us to decrease this time significantly. Considering that the tool may be run once per hour, we believe that the amount of time required may be quite negligible and that the gain obtained by using the optimal scheduling can outperform the cost of the use of the tool, when we consider large applications with long stages.

II.3 Conclusion

This first case study has already shown that our approach can allow grid systems to obtain important information and that we have the potential to enhance the performance of grid applications through the use of structured parallel programs and process algebras. One main limitation of this approach is that the user needs to specify a list of mappings from which the tool will select the best one; the tool is not able to automatically generate mappings (and it would not be affordable to compare all mappings, since there is an exponential number of them).

More importantly, some issues are raised by the use of exponential models having memoryless properties: the throughput computed from the PEPA model turns out to be far from the throughput that we would expect (and compute with our analytical formulas for the deterministic case, see Chapter 2). For instance, consider a very simple pipeline application composed of two stages, each of them being mapped on a different processor. Stage 1 can either be processing a data (state 1w) or sending result to the next stage (state 1c). Similarly, Stage 2 alternates between a communication state (state 2c) in which it receives data from the first stage, and a working state (state 2w). Let us assume that processing a data takes (in average) one slot of time for both stages, and that communication is very fast. The underlying Markov chain corresponding to the PEPA model is depicted in Figure 4.1, where state (1x,2y) means that stage 1 is in state 1x, and stage 2 in state 2y. Therefore, in the Markov chain, transition from (1w,2w) to (1c,2c) takes two transitions which corresponds to two time slots. In fact, when a transition is fired, the system "forgets" that the other stage had been working as well, and processing must start from the beginning again. The expected behavior would be that transition from (1c,2w) or (1w,2c) to (1c,2c) is much shorter because we come from state (1w,2w) in which both

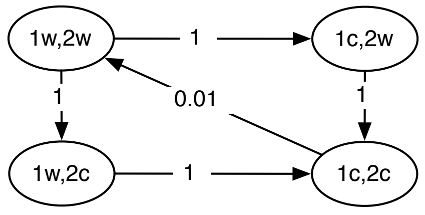


Figure 4.1: Markov chain for a 2-stage pipeline.

stages were working in parallel. However, by definition of the Markov property, this transition is independent from previous state, there is no memory of what happened earlier in the system. If we study the stationary solution of this Markov chain, we obtained the fact that 1/3 of the time is spent on each of the three states (1w,2w), (1c,2w) and (1w,2c). The transition leaving (1c,2c) is very fast, thus the average time spent in this state is almost null. This means that during 1/3 of the time, stage 1 is idle and waits until stage 2 is ready to communicate. The same occurs for stage 2. In other words, the throughput of this pipeline is 2/3, while we would expect a throughput of 1 in a deterministic fully parallel system.

We therefore decided to investigate non-markovian models in order to accurately capture performance. A model based on timed Petri nets is presented in the following section.

III Timed Petri nets

In this section, we investigate the use of timed Petri nets (TPNs), as defined in [11], in order to compute the throughput of the mapping of a linear chain application with replication. Indeed, the period is dictated by the critical hardware resource when stages are not replicated (i.e., assigned to a single processor), but the problem gets surprisingly complicated with replication, and we provide examples where the optimal period is larger than the largest cycle-time of any resource. In Section III.1, we exhibit timed Petri net models under the one-port model, either with or without overlap. Thanks to these models, we are able to compute the optimal throughput (period) of a given mapping, as explained in Section III.2. Finally we conclude in Section III.3.

III.1 Timed Petri net models

We aim in this section at deriving a timed Petri net model to represent a mapping of a linear chain application with replication. Also, we restrict here to one-to-one mappings, and thus a processor can process at most one stage (even though a stage may be processed by several processors, in round-robin fashion).

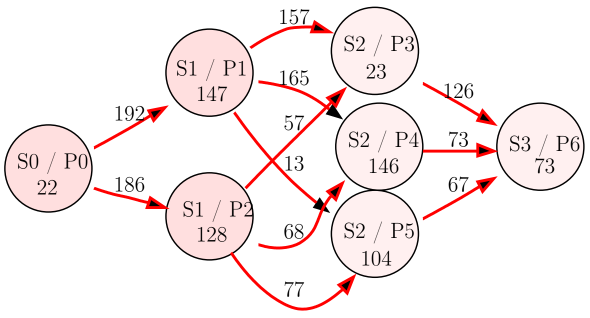


Figure 4.2: Example A: Mapping with replication: S_1 on 2 processors, S_2 on 3 processors.

Input data	Path in the system
0	$P_0 \to P_1 \to P_3 \to P_6$
1	$P_0 \to P_2 \to P_4 \to P_6$
2	$P_0 \to P_1 \to P_5 \to P_6$
3	$P_0 \to P_2 \to P_3 \to P_6$
4	$P_0 \to P_1 \to P_4 \to P_6$
5	$P_0 \to P_2 \to P_5 \to P_6$
6	$P_0 \to P_1 \to P_3 \to P_6$
7	$P_0 \to P_2 \to P_4 \to P_6$

Table 4.1: Example A: Paths followed by the first input data.

In the following only TPNs with the *event graph property* (each place has exactly one input and one output transition) are considered (see [12]). Also, in all our Petri net models, the use of a physical resource during a time t (i.e., the computation of a stage or the transmission of a file from a processor to another one) is represented by a transition with a firing time t, and dependencies are represented using places. Now, let us focus on the path followed in the pipeline by a single input data set, for a mapping with several stages replicated on different processors. Consider Example A described in Figure 4.2: the first data set enters the system and proceeds through processors P_0, P_1, P_3 and P_6. The second data set is first processed by processor P_0, then by processor P_2 (even if P_1 is available), by processor P_4 and finally by processor P_6. Paths followed by the first eight input data sets are summarized up in Table 4.1: as we can see, there are 6 different paths followed by the data sets, and then data set i takes the same path as data set $i-6$. We have the following easy result (see [C34] for a proof):

Proposition 1. *Consider a pipeline of n stages S_0, ..., S_{n-1}, such that stage S_i is mapped onto m_i distinct processors. Then the number of paths followed by the input data in the whole system is equal to $m = \operatorname{lcm}(m_0, \ldots, m_{n-1})$.*

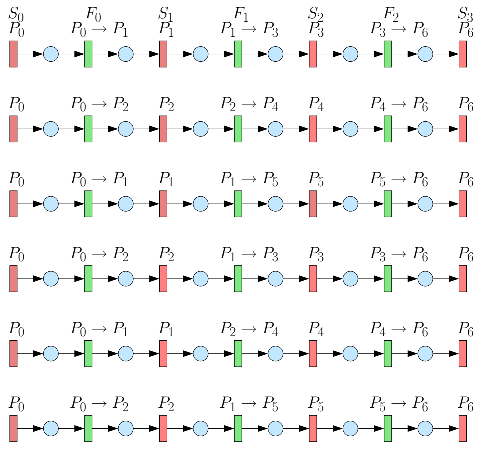
The TPN model given here is inspired from the TPN model of jobshops with static schedules from [61]. Here, however, replication imposes that each path followed by the input data must be fully developed in the TPN: if P_0 appears in several distinct paths, as in the example of Figure 4.2, then there are several transitions corresponding to P_0 in the TPN. Furthermore, we have to add dependencies between all the transitions corresponding to the same physical resource to avoid the simultaneous use of the same resource by different input data. These dependencies differ according to the model used for communications and computations (with or without overlap).

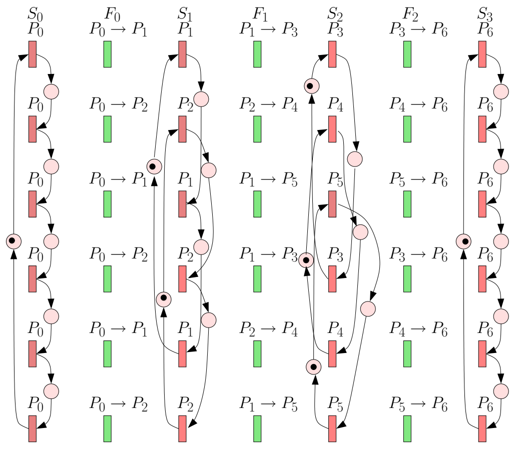
Model with overlap. First, let us focus on the model with overlap: any processor can receive a file and send another one while computing. All paths followed by the input data in the whole system have to appear in the TPN. We use the notations of Proposition 1.

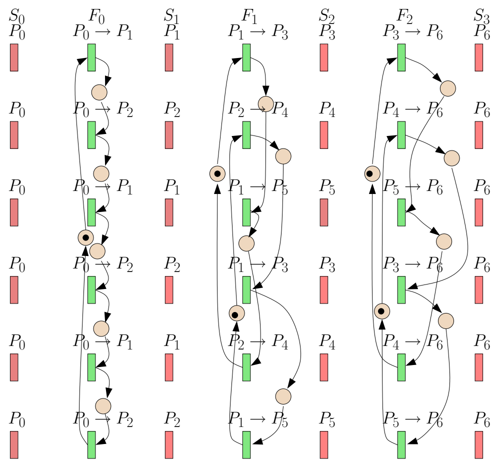
Let m denote the number of paths of our mapping. Then the i-th input data follows the $(i \mod m)$-th path, and we have a rectangular TPN, with m rows of $2n-1$ transitions, due to the n transitions representing the use of processors and the $n-1$ transitions representing the use of communication links. The i-th transition of the j-th row is named T_i^j. The time required to fire a transition T_{2i}^j, corresponding to the processing of stage S_i (mapped on P_u), is set to $\frac{w_i}{s_u}$. The time required by a transition T_{2i+1}^j, corresponding the transmission of a file from S_i (mapped on P_u) to S_{i+1} (mapped on P_v), is set to $\frac{\delta_i}{b_{u,v}}$.

Then, we need to add places between these transitions, in order to model the following set of constraints:

1. The file F_i cannot be sent before the end of the computation of S_i: a place is added from T_{2i}^j to T_{2i+1}^j on each row. Similarly, the stage S_{i+1} cannot be processed before the end of the communication of F_i: a place is added from T_{2i+1}^j to $T_{2(i+1)}^j$ on each row j. All these places are shown in Figure 4.3(a).
2. When a processor appears in several rows, the round-robin distribution imposes dependencies between these rows. Assume that processor P_i appears on rows $j_1, j_2, \ldots, j_k$. Then we add a place from $T_{2i}^{j_l}$ to $T_{2i}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $T_{2i}^{j_k}$ to $T_{2i}^{j_1}$. All these places are shown in Figure 4.3(b).
3. The one-port model and the round-robin distribution of communications also impose dependencies between rows. Assume that processor P_i appears on rows $j_1, j_2, \ldots, j_k$. Then we add a place from $T_{2i+1}^{j_l}$ to $T_{2i+1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $T_{2i+1}^{j_k}$ to $T_{2i+1}^{j_1}$ to ensure that P_i does not send two files simultaneously, if P_i does not compute the last stage. All these places are shown in Figure 4.3(c).
4. In the same way, we add a place from $T_{2i-1}^{j_l}$ to $T_{2i-1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $T_{2i-1}^{j_k}$ to $T_{2i-1}^{j_1}$ to ensure that P_i does not receive two files simultaneously, if P_i does not compute the first stage. All these places are shown in Figure 4.3(d).


(a) Dependencies between communications and computations.


(b) Dependencies due to the round-robin distribution of computations.


(c) Dependencies due to the round-robin distribution of outgoing communications.

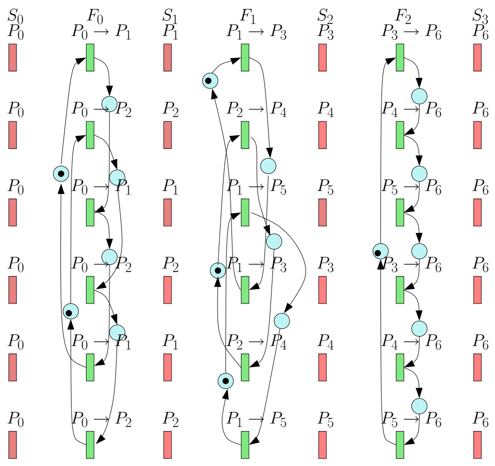

(d) Dependencies due to the round-robin distribution of incoming communications.

Figure 4.3: With overlap: places imposed by the different constraints. Circuits model the round-robin distribution, and the single token in each circuit models the fact that any resource can process at most one job at a time.

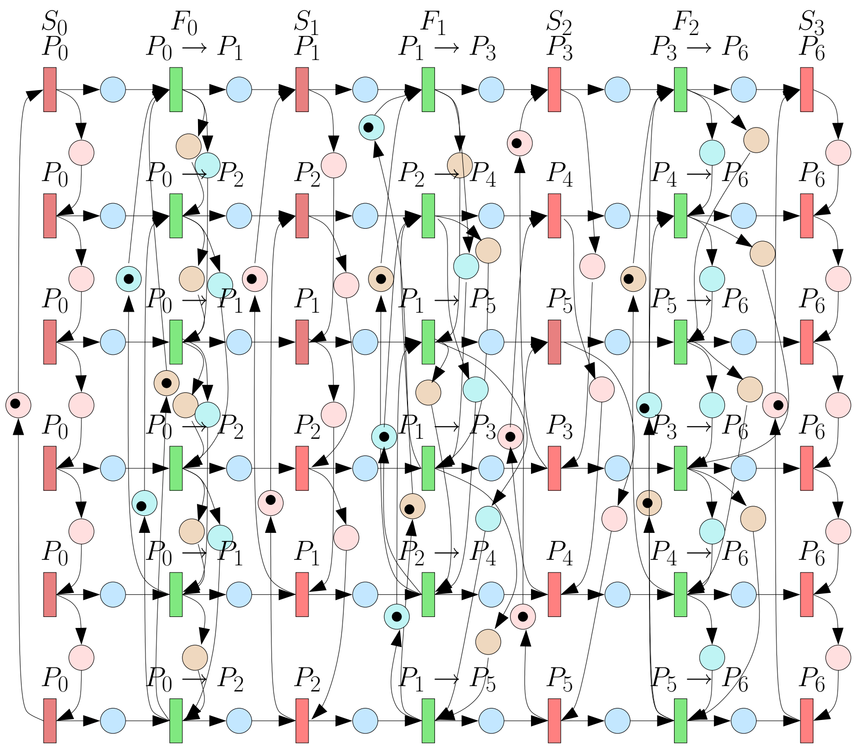
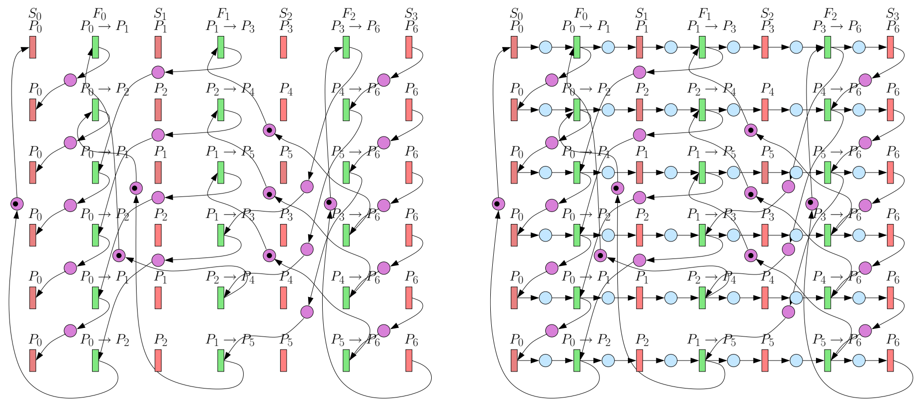


Figure 4.4: Complete TPN of Example A for the model with overlap.

Finally, any resource before its first use is ready to compute or communicate, only waiting for the input file. Indeed, a token is put in every place going from a transition $T_i^{j_k}$ to a transition $T_i^{j_1}$, as defined in the previous lines. The complete TPN of Example A for the model with overlap is given in Figure 4.4.

Model without overlap. In the model without overlap, any processor can either send a file, receive another one, or perform a computation while these operations were happening concurrently in the model with overlap. Hence, we require a processor to successively receive the data from stage S_{i-1}, compute the stage S_i and send the result to S_{i+1} before receiving the next data set of S_{i-1}. Paths followed by the input data are obviously the same as in the case with overlap, and the structure of the TPN remains the same (m rows of $2n-1$ transitions).

The first set of constraints is also identical to that of the model with overlap, since we still have dependencies between communications and computations, as in Figure 4.3(a). However, the other dependencies are replaced by those imposed by the round-robin order of the model without overlap. Indeed, when a processor appears in several rows, the round-robin order imposes dependencies between these rows. Assume that processor P_i appears on rows $j_1, j_2, \ldots, j_k$. Then we add a place from $T_{2i+1}^{j_l}$ to $T_{2i-1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $T_{2i+1}^{j_k}$ to $T_{2i-1}^{j_1}$. These places ensure



(a) Dependencies due to the round-robin distribution.

(b) Complete TPN.

Figure 4.5: TPN of Example A for the model without overlap.

the respect of the model: the next reception cannot start before the completion of the current sequence reception-computation-sending. All these places are shown in Figure 4.5(a).

Any physical resource can immediately start its first communication, since it is initially only waiting for the input communication. Thus a token is put in every place from a transition $T_i^{j_k}$ to a transition $T_i^{j_1}$, as defined in the previous lines. The complete TPN of Example A for the model without overlap is given in Figure 4.5(b).

The automatic construction of the TPN in both cases has been implemented. The time needed to construct the Petri net is linear in its size: $O(mn)$.

III.2 Computing the throughput of a mapping

TPNs with the event graph property make the computation of the throughput of a complex system possible through the computation of *critical cycles*, using $(\max, +)$ algebra [12]. For any cycle $\mathcal{C}$ in the TPN, let $\mathcal{L}(\mathcal{C})$ be its length (number of transitions) and $t(\mathcal{C})$ be the total number of tokens in places traversed by $\mathcal{C}$. Then a critical cycle achieves the largest ratio over all cycles, $\max_{\mathcal{C}} \frac{\mathcal{L}(\mathcal{C})}{t(\mathcal{C})}$, and this ratio is the period $\mathcal{P}$ of the system: indeed, after a transitive period, every transition of the TPN is fired exactly once during a period of length $\mathcal{P}$ [12].

Critical cycles can be computed with softwares like ERS [72] or GreatSPN [36] with a complexity $O(m^3n^3)$. By definition of the TPN, the firing of any transition of the last column corresponds to the completion of the last stage, i.e., to the completion of an instance of the workflow. Moreover, we know that the m transitions of this last column (where m is still the number of rows of the TPN) are fired in a round-robin order. In our case, m data sets are completed during any period $\mathcal{P}$: the obtained throughput is $\frac{m}{\mathcal{P}}$.

Model with overlap. The TPN associated to the model with overlap has a regular structure, which facilitates the determination of critical cycles. In the complete TPN, places are linked to transitions either in the same row and oriented forward, or in the same column. Hence, each cycle only contains transitions belonging to the same column: we can split the complete TPN into $2n-1$ smaller TPNs, each sub-TPN representing either a communication or a computation. However, the size of each sub-TPN (the restriction of the TPN to a single column) is not necessarily polynomial in the size of the instance, due to the possibly large number of rows, equal to $m = \mathrm{lcm}\,(m_0, \ldots, m_{n-1})$.

It turns out that a polynomial algorithm exists to find the weight $\mathcal{L}(\mathcal{C})/t(\mathcal{C})$ of a critical cycle: only a fraction of each sub-TPN is required to compute this weight, without computing the cycle itself. The algorithm and its proof are very involved and can be found in [C34].

Theorem 8. *Consider a pipeline of n stages $S_0, \ldots, S_{n-1}$, such that stage S_i is mapped onto m_i distinct processors. Then the average throughput of this system can be computed in time $O\left(\sum_{i=0}^{n-2}(m_i m_{i+1})^3\right)$.*

In Example A, a critical resource is the output port of P_0, whose cycle-time is equal to the period, 189. However it is possible to exhibit cases without critical resources: see for instance Example B presented in Figure 4.6. The cycle-time of a critical resource is 258.3, and it corresponds to the outgoing communications of P_2. This value is strictly smaller than the actual period of the complete system (291.7).

Model without overlap. Cycles in the TPN associated to the model without overlap are more complex and less regular, since corresponding TPNs have backward edges. An example of such a cycle is shown in Figure 4.7. The intuition behind these backward edges is that a processor P_u cannot compute a data set of S_i before having completely sent the result of the previous data set to the next processor P_v in charge of S_{i+1}. Thus, P_u can be slowed by P_v. As for the model with overlap, there exist mappings for which all resources have idle times during a complete period. For the model without overlap, this is the case of Example A. The corresponding Gantt diagram is presented in Figure 4.8: the critical resource is P_2, which has a cycle-time of 215.8, strictly smaller than the period (230.7).

III.3 Conclusion

In this section, we showed how TPNs (timed Petri nets) are a useful model in order to determine the critical cycles, and thus the optimal throughput, of a given mapping with replication. The complexity of throughput evaluation depends on the communication model. Even the simple round-robin distribution implies complex interactions between involved resources, resulting in schedules without any critical resource: there exist schedules, such that all resources remain partially idle, and this is true both for a model with overlap and a model without overlap. We have also

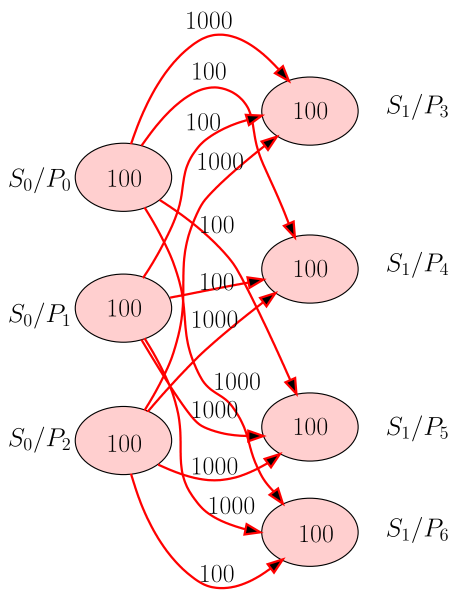


Figure 4.6: Example B: Stage 0 is replicated on 3 processors, and Stage 1 on 4 processors.

conducted experiments (see [C34]), in order to find out how often such cases occur. On all mappings that were randomly generated and tested, we realized that such cases are very rare under the model with overlap. In addition, we have established the polynomial complexity of the problem for this model with overlap, while it turns out to be NP-hard for the model without overlap[3]. For both cases, the TPN model allows us to compute the period, thanks to the tool which computes the critical cycles of a TPN.

Compared to the approach of Section II, we are back to a fully deterministic setting in the current section, and thus we do not target dynamic platforms, as motivated earlier. Also, we restrict to one-to-one mappings. However, thanks to this non-markovian model, we were able to compute the throughput of a given mapping, and identify cases in which there were no critical resources. Moreover, we believe that this study opens a way on finding good schedules on dynamic platforms, whose speeds and bandwidths are modeled by random variables. Indeed, it is possible to introduce random laws in the timed Petri net models, in order to replace the deterministic firing times.

In the next section, we do not account for platform variability either, but we consider dynamic platforms subject to failures. However, we move to a simpler application setting, in which a single divisible workload must be processed.

[3]This complexity result has been established recently with Fanny Dufossé, Matthieu Gallet and Yves Robert. It is not yet published, but please ask me for details.

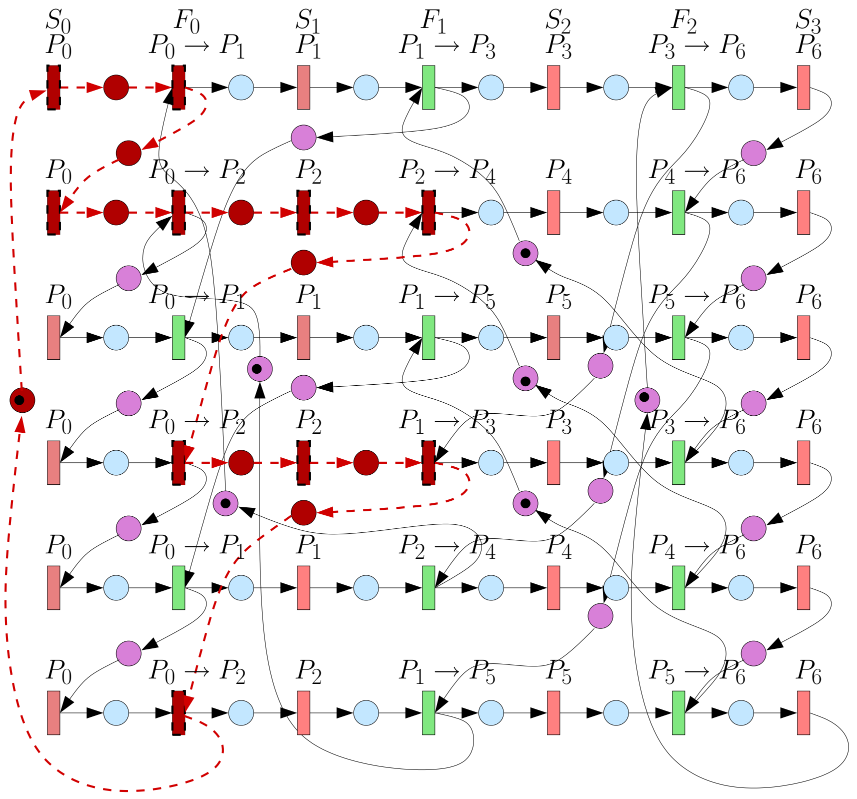


Figure 4.7: Complex critical cycles on Example A.

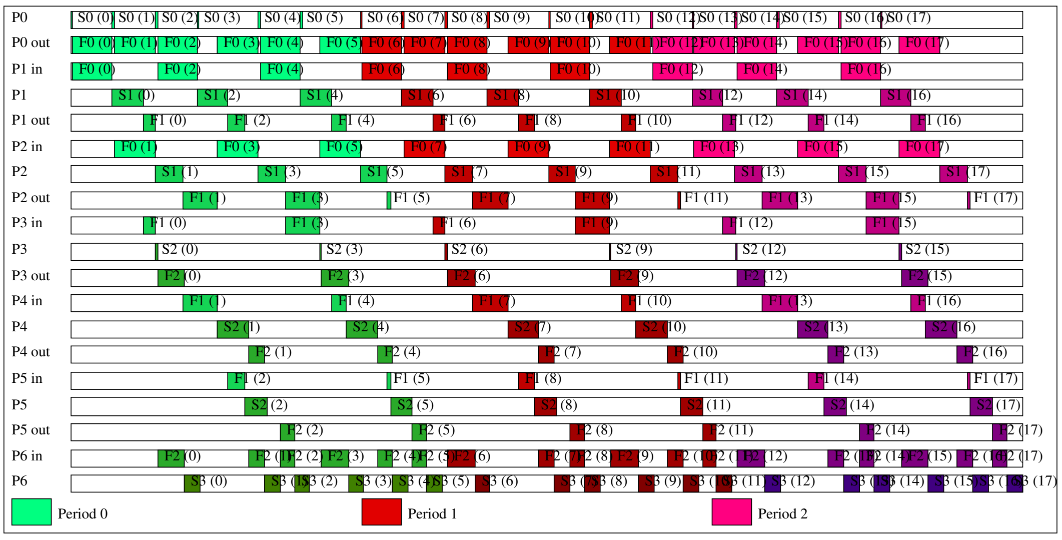


Figure 4.8: Gantt diagram of a schedule without critical resource.

IV Divisible workload

In this section, we follow in the steps of sources such as [7, 24, 107], and we propose an analytic study of algorithmic techniques for coping with uncertainty in computational settings. Whereas most of the previous work addresses the uncertainty for an application executed on a single computer, we consider here a set of computers, such as in a Grid computing [51] or volunteer computing [83] environment. Therefore, we assume that one computer's shortcomings can be compensated for by other computers, most notably by judiciously replicating work, i.e., allocating some work to more than one computer.

However, unlike previous chapters targeting pipelined applications and with a failure probability independent of time, we target here a large computational workload whose constituent work is *divisible* in the sense that each chunk of work can be partitioned into arbitrary granularity (see [21]). The similar point is that we consider unrecoverable interruptions that cause us to lose all work currently in progress on the interrupted computer. We have access to $p \geq 1$ identical computers[4] to compute the workload via worksharing. The only external resource to help us use this tool judiciously is our assumed access to *a priori* knowledge of the risk of a computer's having been interrupted—which we assume is the same for all computers (as in [24, 107], our scheduling strategies can be adapted to use statistical, rather than exact, knowledge of the risk of interruption—albeit at the cost of weakened performance guarantees). Most of our results assume the linear risk model, in which the probability that a computer will be interrupted increases linearly with the time the computer has been available.

Our goal is then to maximize the expected amount of work that gets computed, no matter which, or how many computers get interrupted. Therefore, we implicitly assume that we are dealing with applications for which even partial output is meaningful, e.g., Monte-Carlo simulations.

We first describe the challenges that arise from such application in Section IV.1. Then we expose the technical framework in Section IV.2 and in particular we introduce two models, either with or without initiation cost. We give a selection of results for the simple cases with one or two processors in Section IV.3, and discuss more complex cases in the conclusion in Section IV.4. The details of this work can be found in [C25], and proofs have therefore been omitted in the following.

IV.1 The challenges

The challenges of scheduling a divisible workload on interruptible remote computers can be described in terms of dilemmas.

First, we note that sending each remote computer a small amount of work minimizes vulnerability to interruption-induced losses, but it maximizes the impact of per-work overhead, and thus minimizes the opportunities for parallelism. We cope with this first dilemma by sending work allocations to computers as a sequence

[4]The study of a heterogeneous platform is kept for future work.

of chunks rather than as a single block per computer. This approach, advocated in [24, 107], allows each computer to checkpoint at various times, thereby, protecting some work from the threat of interruption. One challenge consists in finding which size of chunks should be used.

Also, replicating work lessens vulnerability to interruption-induced losses, but it minimizes the expected productivity advantage from having access to remote computers (the pros and cons of work replication are discussed in [82]). We thus may consider to allocate chunks to more than one remote computer in order to enhance their chances of being computed successfully. Work should be replicated judiciously, in deference to this second dilemma.

Because communication to remote computers is likely to be costly in time and overhead, we limit such communication by orchestrating work replications in a static manner, rather than dynamically, in response to observed interruptions. While we thereby duplicate work unnecessarily when there are few interruptions among the remote computers, we also prevent the server from becoming a communication bottleneck.

IV.2 The framework

We have W units of divisible work to execute onto $p \geq 1$ identical computers that are susceptible to unrecoverable interruptions that kill all work in progress. All computers share the same perfectly known instantaneous probability of being interrupted, and this probability increases with the amount of time the computer has been operating (whether working or not). As discussed earlier, the danger of losing work in progress when an interruption incurs mandates that we not just divide the workload into W/p equal-size chunks and allocate one chunk to each computer. Instead, we aim at: (i) partitioning the workload into chunks, the unit of work that we allocate to the computers; (ii) prescribing a schedule for allocating chunks to computers; and (iii) allocating some chunks to several computers, as a divisible-load analogue of task replication.

Within this model, all computers share the same risk function, i.e., the same instantaneous probability, $Pr(w)$, of having been interrupted by the end of the first w time units. We measure time in terms of work units that could have been executed successfully, i.e., with no interruption. In other words "the first w time units" is the amount of time that a computer would have needed to compute w work units if it had started working on them when the entire worksharing episode began. This time scale is shared by all computers. Of course, $Pr(w)$ increases with w; moreover, we assume that we know its exact value.

It is useful in our study to generalize our measure of risk by allowing one to consider many baseline moments. We denote by $Pr(s, w)$ the conditional probability that a remote computer will be interrupted during the next w time units, given that it has not been interrupted during the first s time units. Thus, $Pr(w) = Pr(0, w)$ and $Pr(s, w) = Pr(s + w) - Pr(s)$. We let $\kappa \in (0, 1]$ be a constant that weights our probabilities.

The risk function that is the focus of this study is the linear function $Pr(w) = \kappa w$. It is the most natural model in the absence of further information: the risk of interruption grows linearly with the time that the computer has been available, or equivalently to the amount of work it could have done. The density function is then $dPr = \kappa dt$ for $t \in [0, 1/\kappa]$ and 0 otherwise, so that

$$Pr(s,w) \ = \ \min\left\{1, \ \int_s^{s+w} \kappa dt\right\} \ = \ \min\{1, \kappa w\}.$$

The constant $1/\kappa$ can be viewed as the time by which an interruption will have occurred with probability 1, and is also denoted by X.

Risk functions help us finding an efficient way to chunk work for, and allocate work to, the remote computers, in order to maximize the expected work production of the assemblage. Let jobdone be the random variable whose value is the number of work units that the assemblage executes successfully under a given scheduling regimen. Formally, we are striving to maximize the expected value (or, expectation) of jobdone.

We perform our study under two models, which play different roles as one contemplates the problem of scheduling a large workload. The models differ in the way chunk execution times relate to chunk sizes. In short, there are two classes of time-costs, those that are proportional to the chunk size and those that are fixed constants. When chunks are large, the second cost will be minuscule compared to the first. This suggests that the fixed costs can be ignored, but one must be careful: if one ignores the fixed costs, then there is no disincentive to, say, deploying the workload to the remote computers in $n + 1$ chunks, rather than n. Of course, increasing the number of chunks tends to make chunks smaller—which increases the significance of the fixed cost! Therefore, we perform the current study with two cost models, striving for optimal schedules under each one. The *free-initiation model* is characterized by *not* charging the owner of the workload a per-chunk fixed cost. This model focuses on situations wherein the fixed costs are negligible compared to the chunk-size-dependent costs. The *charged-initiation model*, which more accurately reflects the costs incurred with real computing systems, is characterized by accounting for both the fixed and chunk-dependent costs.

The *free-initiation* model. This model assesses no per-chunk cost. Our results under this model approximate reality well when *a priori* chunks must be large, e.g., because large fixed-size costs demand that every remote computer do a substantial amount of work in order to amortize the fixed-size costs. In such a situation, one keeps chunks large by placing a bound on the number of scheduling "rounds," which counteracts this model's tendency to increase the number of "rounds" without bound. Importantly also: the free-initiation model allows us to obtain *bounds* on the expectation of jobdone under the charged-initiation model, when such bounds are prohibitively hard to derive directly (see Theorem 9).

Under the free-initiation model, the expected value of jobdone under a given scheduling regimen Θ, denoted $E^{(\mathrm{f})}(\mathsf{jobdone}, \Theta)$, the superscript "f" denoting "free-initiation," is

$$E^{(\mathrm{f})}(\mathsf{jobdone}, \Theta) \;=\; \int_0^{\top} Pr(\mathsf{jobdone} \geq u \text{ under } \Theta)\; du.$$

For instance, if the workload is deployed as a single chunk, we have $E^{(\mathrm{f})}(W, \Theta_1) = W\,(1 - Pr(W))$, while for two chunks,

$$\begin{aligned} E^{(\mathrm{f})}(W, \Theta_2) \;&=\; \int_0^{\omega_1} Pr(\mathsf{jobdone} \geq u) du + \int_{\omega_1}^{\omega_1+\omega_2} Pr(\mathsf{jobdone} \geq u) du \\ &=\; \omega_1(1 - Pr(\omega_1)) \;+\; \omega_2(1 - Pr(\omega_1 + \omega_2)). \end{aligned}$$

The *charged-initiation* model. This model is much harder to analyze than the free-initiation model, even when there is only one remote computer. In compensation, this model often allows one to determine analytically what the best numbers of chunks are. Under this model, the overhead for each additional chunk is a fixed cost—which, in common with time, we measure in units of work—that is added to the cost of computing of each chunk; we denote this overhead by ε (for instance this may correspond to a checkpointing cost). Under this model, then, the expectation of $\mathsf{jobdone}$ under schedule Θ, denoted $E^{(\mathrm{c})}(\mathsf{jobdone}, \Theta)$, the superscript "c" denoting "charged(-initiation)," is

$$E^{(\mathrm{c})}(\mathsf{jobdone}, \Theta) \;=\; \int_0^{\top} Pr(\mathsf{jobdone} \geq u + \varepsilon)\; du.$$

When the whole workload is deployed as a single chunk, $E^{(\mathrm{c})}(W, \Theta_1) \;=\; W(1 - Pr(W + \varepsilon))$, and when work is deployed as two chunks of respective sizes ω_1 and ω_2, $E^{(\mathrm{c})}(W, \Theta_2) \;= \omega_1(1 - Pr(\omega_1 + \varepsilon)) + \omega_2(1 - Pr(\omega_1 + \omega_2 + 2\varepsilon))$.

Relating the two models. One can bound the work completed under the charged-initiation model via the free-initiation model. This justifies the study of the free-initiation model.

Theorem 9. *Let $E^{(\mathrm{c})}(W, n)$ and $E^{(\mathrm{f})}(W, n)$ denote, respectively, the optimal expected value of* $\mathsf{jobdone}$ *with n chunks under the* charged-*initiation model and under the* free-*initiation model. Then:*

$$E^{(\mathrm{f})}(W, n) \;\geq\; E^{(\mathrm{c})}(W, n) \;\geq\; E^{(\mathrm{f})}(W, n) - n\varepsilon.$$

IV.3 With one or two computers

In this section, we only present results on how to schedule under the linear risk model in a simple case with one or two remote computers, and the extension will be briefly discussed in Section IV.4. Some of the results we derive bear a striking similarity to their analogues in [24], despite certain substantive differences in models.

IV.3.1 With one computer

We first illustrate why the risk of losing work because of an interruption must affect our scheduling strategy, even when there is only one remote computer and under the free-initiation model. When $W \leq 1/\kappa$, the expected amount of work achieved when one deploys the entire workload in a single chunk is $E^{(\mathrm{f})}(W, \Theta_1) = W - \kappa W^2$. The analogous quantity when one deploys the workload in two chunks, of respective sizes $\omega_1 > 0$ and $\omega_2 > 0$, with $\omega_1 + \omega_2 = W$, is $E^{(\mathrm{f})}(W, \Theta_2) = W - W^2\kappa + \omega_1\omega_2\kappa$. Note that: $E^{(\mathrm{f})}(W, \Theta_2) - E^{(\mathrm{f})}(W, \Theta_1) = \omega_1\omega_2\kappa > 0$. Thus, as one would expect: *For any fixed total workload, one increases the expectation of* jobdone *by deploying the workload as two chunks, rather than one—no matter how one sizes the chunks.* In fact, the expectation of jobdone for the optimal schedule strictly increases with the number of chunks allowed (Theorem 10). This fact identifies a weakness of the free-initiation model: increasing the number of chunks always increases the expected amount of work done—so the (unachievable) “optimal” strategy would deploy infinitely many infinitely small chunks.

Theorem 10. (free-initiation model) *Say that one wishes to deploy W units of work to a single remote computer in at most n chunks, for some integer $n > 0$. In order to maximize the expectation of* jobdone*, one should have all chunks comprise Z/n units of work, where $Z = \min\left\{W, \frac{n}{n+1}X\right\}$. In expectation, this optimal schedule completes $E^{(\mathrm{f})}(W, n) = Z - \frac{n+1}{2n}Z^2\kappa$ units of work.*

The *charged-initiation* analogue of Theorem 10 is dramatically more difficult to deal with.

Theorem 11. (charged-initiation model) *Say that one wishes to deploy W units of work to a single remote computer in at most n chunks, for some integer $n > 0$; say that $X \geq \varepsilon$. Let $n_1 = \left\lfloor \frac{1}{2}\left(\sqrt{1 + 8X/\varepsilon} - 1\right)\right\rfloor$ and $n_2 = \left\lfloor \frac{1}{2}\left(\sqrt{1 + 8W/\varepsilon} + 1\right)\right\rfloor$. The unique regimen for maximizing $E^{(\mathrm{c})}$(jobdone) specifies $m = \min\{n, n_1, n_2\}$ chunks: the first has size $\omega_{1,m} = \frac{Z}{m} + \frac{m-1}{2}\varepsilon$, where $Z = \min\left\{W, \frac{m}{m+1}X - \frac{m}{2}\varepsilon\right\}$; the $(i+1)$th chunk inductively has size $\omega_{i+1,m} = \omega_{i,m} - \varepsilon$. In expectation, this schedule completes $E^{(\mathrm{c})}(W, n) = Z - \frac{m+1}{2m}Z^2\kappa - \frac{m+1}{2}Z\varepsilon\kappa + \frac{(m-1)m(m+1)}{24}\varepsilon^2\kappa$ units of work.*

The proofs, which are very technical and difficult, especially under the charged-initiation model, can be found in [C25].

IV.3.2 With two computers

Results in [C25] suggest that finding exactly optimal schedules is surprisingly difficult as soon as we consider two computers. However, we were able to establish characteristics of optimal schedules under general risk functions. We consider two remote computers, P_1 and P_2, under the free-initiation model.

Say, for $i = 1, 2$, that we deploy n_i chunks of work, $\mathcal{W}_{i,1}, \ldots, \mathcal{W}_{i,n_i}$, on P_i; P_i must schedule them in this order. We do not assume any *a priori* relation between

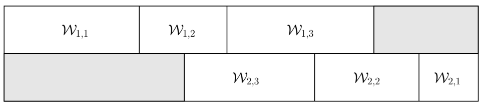


Figure 4.9: The shape of Theorem 12's optimal schedule for two computers; $n_1 = n_2 = 3$. The top row displays P_1's chunks, the bottom row P_2's. Vertically aligned parts of chunks correspond to shared work; shaded areas depict unallocated work (e.g., no work in $\mathcal{W}_{2,1}$ is allocated to P_1).

how P_1 and P_2 break their allocated work into chunks: work that is allocated to both P_1 and P_2 may be chunked differently on the two machines. The proof of the following theorem can be found in [C25].

Theorem 12. *Let Θ be a schedule for P_1 and P_2. Say that, for both computers, the probability of being interrupted never decreases as a computer processes more work. Then there exists a schedule Θ' that, in expectation, completes as much work as does Θ and that satisfies the following three properties; cf. Fig. 4.9.*

*1. **Maximal work deployment.** Θ' deploys as much of the overall workload as possible: the workloads it deploys to P_1 and P_2 can overlap only if their union is the entire overall workload.*

*2. **Local work priority.** Θ' has P_1 (resp., P_2) process all of the allocated work that it does* not *share with P_2 (resp., P_1) before it processes any shared work.*

*3. **Shared work "mirroring."** Θ' has P_1 and P_2 process their shared work "in opposite orders." Specifically, say for $k = 1, 2$, that P_k chops its allocated work into chunks $\mathcal{W}_{k,1}, \ldots, \mathcal{W}_{k,n_k}$. Say that there exist chunk-indices a_1, $b_1 > a_1$ for P_1, and a_2, $b_2 > a_2$ for P_2 such that: $\mathcal{W}_{1,a_1}$ and $\mathcal{W}_{2,a_2}$ both contain a shared "piece of work" A, and $\mathcal{W}_{1,b_1}$ and $\mathcal{W}_{2,b_2}$ both contain a shared "piece of work" B. Then if Θ' has P_1 execute A before B (i.e., P_1 executes $\mathcal{W}_{1,a_1}$ before $\mathcal{W}_{1,b_1}$), then Θ' has P_2 execute B before A (i.e., P_2 executes $\mathcal{W}_{2,b_2}$ before $\mathcal{W}_{2,a_2}$).*

IV.4 Conclusion

We have presented in this section a selection of results from [C25], and demonstrated how complicated the problem can become in a dynamic setting with unrecoverable interruptions. Indeed, we were able to optimally solve the problem only when considering a single computer, and we characterized the shape of an optimal solution with two computers. The complexity of the development about two computers suggests that the general case of p remote computers will be prohibitively difficult, even with respect to deriving asymptotically optimal schedules. For this general case, we derived a number of well-structured heuristics, whose quality could be assessed via explicit expressions for their expected work outputs. Extensive simulation experiments suggest that our heuristics provide schedules with good expected work output in non trivial cases, that is, when there is work to replicate ($W < pX$) and the replication is not trivial ($W > X$).

Much remains to be done regarding this important problem, but three directions stand out as perhaps the major outstanding challenges. One of these is to extend the (asymptotic-)optimality results to a larger class of risk functions, thereby covering the range of situations that this work addresses. A second is to extend this study to heterogeneous platforms, whose constituent computers differ in speed and other computational resources. Finally, it would be significant to allow computers to be subject to differing probabilities of being interrupted.

We compare the approach of this section with the other works presented so far in this document in the following Section V.

V Summary and conclusion

In this chapter, we have demonstrated the additional difficulties raised when addressing a dynamic setting, with platform variability and failures. Already, we were confronted to many problems in a static framework such as the one exposed in Chapter 2, and the use of performance models allowed us to solve some of the previous issues. Thus, in Section III, we were able to compute the period of a one-to-one mapping with replication, either with a polynomial algorithm for the model with overlap, or with the help of a tool which computes the critical cycles of a TPN for the model with overlap (for which the problem is NP-hard). This section was not addressing variability yet, contrary to Section II which considered a platform with parameters following an exponential law. However, the memoryless property of Markov chains made us unable to obtain an accurate performance prediction from this first model, hence the modeling of Section III. Of course, much work remains to be done in order to transform the latter model in a non-deterministic one, to account for platform variability. We are currently working on this point.

A dynamic platform is characterized by its variability, as we targeted in Sections II and III. However, a dynamic platform also is subject to failures, and we focused on the failure model in Section IV. In the linear chain pipelined application framework that we studied so far, we motivated a constant failure model. However, such a model has its limitations, and may not always be realistic. We thus investigated a failure risk which increases with time, but left the linear chain framework for a simpler one, a single divisible workload to be computed by a set of homogeneous processors. Even in this simple framework, the problems turn out very complicated, and we were not even able to derive optimal results with two identical processors. This section fully demonstrated the complexity of realistic failure models.

We have conducted other work which deal with failures. In the context of DAG scheduling, we have investigated task replication in order to be able to tolerate a fixed number of failures, see for instance [J11],[J15]. In a different context of scheduling for a micro-factory, we studied the problem with task failures rather than hardware failures (see [C30]). Our current research activity focuses on dynamic platforms, and we hope to be able to solve some of the challenges raised by the study of such platforms.

We have so far concentrated the study to linear chain applications, or even simpler application such as the divisible workload of Section IV. In the next chapter, we discuss problems raised by more complex applications.

CHAPTER 5

Beyond linear chains

We have extensively studied linear chain applications, both in a static and a dynamic setting, and the conclusion is that even for simple application patterns, problems can get incredibly difficult. In this chapter, we target more complex applications, and exhibit cases in which we are able to derive interesting complexity results.

First in Section I, we add a selectivity parameter to each application stage, hence expressing the fact that a stage can filter or expand the data. This leads to add dependencies between stages in order to exploit this feature, even if there are no dependence constraints initially. In Section II, we tackle applications with more complex dependencies, such as fork graphs and directed acyclic graphs (DAGs), still in the context of pipelined applications. Then Sections III and IV study more complex and more realistic application scenarios. The first one is the problem of replica placement in tree networks, which aims at satisfying requests coming from clients, while optimizing some objective function. The second one considers applications which are structured as trees of operators: basic objects must be downloaded (leaves of the tree) and then operators must be applied continuously, in order to produce results at some desired rate.

I Filtering applications

This section is devoted to the problem of mapping filtering services on large-scale platforms, which corresponds to the problem of query optimization over Web services [114, 28]. This problem is close to the problem of mapping pipelined applications onto distributed architectures, as discussed earlier, but it involves several additional difficulties due to the filtering properties of the services.

Consider a collection of various services that must be applied on a stream of consecutive data sets. As for pipelined applications, we have a graph with nodes (the services) and precedence edges (dependence constraints between services), with data flowing continuously from the input node(s) to the output node(s). Also, the goal is to map each service onto a processor, or server, so as to optimize the same performance objectives as before (period and/or latency). But in addition, services can filter the data by a certain amount, according to their selectivity.

We first expose the new framework and illustrate through examples the additional complexity raised by the filtering properties of stages, even in the absence of communication costs, in Section I.1. Then we present complexity results for such problems in Section I.2. The extension to filtering application with communication costs is discussed in Section I.3. Finally we conclude in Section I.4.

I.1 Filters framework

All hypotheses and mapping rules are those of Srivastava et al. [114, 28]. Although their papers mainly deal with query optimization over Web services (already an increasingly important application with the advent of Web Service Management Systems [49, 99]), the approach applies to general data streams [9] and to database predicate processing [31, 60]. Finally (and quite surprisingly), we note that this framework is quite similar to the problem of scheduling unreliable jobs on parallel machines [3], where service selectivities correspond to job failure probabilities.

The framework. The target application $\mathcal{A}$ is a set of services (or stages, or filters, or queries) linked by precedence constraints. We write $\mathcal{A} = (\mathcal{F}, \mathcal{G})$ where $\mathcal{F} = \{C_1, C_2, \dots, C_n\}$ is the set of services and $\mathcal{G} \subset \mathcal{F} \times \mathcal{F}$ is the set of precedence constraints. If $\mathcal{G} = \{(C_1, C_2), (C_2, C_3), \dots, (C_{n-1}, C_n)\}$, we have a linear chain as before. If $\mathcal{G} = \emptyset$, we have services without precedence constraints. A service C_i is characterized by its cost c_i and its selectivity σ_i.

For the computing resources, we have a set $\mathcal{S} = \{S_1, S_2, \dots, S_p\}$ of servers (or processors). In the case of homogeneous platforms, servers are identical while in the case of heterogeneous platforms, each server S_u is characterized by its speed s_u. We always assume that there are more servers available than services (hence $n \leq p$), and we search a one-to-one mapping[1], or allocation, of services to servers. The one-to-one allocation function *alloc* associates to each service C_i a server $S_{alloc(i)}$. We restrict below to a framework with no communication costs. The extension to communication costs is discussed in Section I.3.

Filtering data. Consider a service C_i with selectivity σ_i: if the incoming data is of size δ, then the outgoing data will be of size $\delta \times \sigma_i$. The initial data is of size δ_0. We see that the data is shrunk by C_i (hence the term "filter") when $\sigma_i < 1$ but it can also be expanded if $\sigma_i > 1$. Each service has an elementary cost c_i, which represents the volume of computations required to process a data set of size δ_0. But the volume of computations is proportional to the actual size of the input data, which may have shrunk or expanded by the predecessors of C_i in the mapping. Altogether, the time to execute a data set of size $\sigma \times \delta_0$ when service C_i is mapped onto server S_u is $\sigma \frac{c_i}{s_u}$. Here σ denotes the combined selectivity of all predecessor of C_i in the mapping.

Execution graph. We also have to build a graph $G = (\mathcal{C}, \mathcal{E})$ that summarizes all precedence relations in the mapping. The nodes of the graph are couples $(C_i, S_{alloc(i)})$, and thus they define the allocation function. There is an arc $(C_i, C_j) \in \mathcal{E}$ if C_i precedes C_j in the execution. There are two types of such arcs: those induced by the set of precedence constraints $\mathcal{G}$, which must be enforced in any case, and those added to reduce the period or the latency. The execution graph G is called a plan.

[1]In general mappings, we can map several services onto the same server. Problems with general mappings are straightforwardly shown NP-hard by reduction from 2-Partition or bin packing [54].

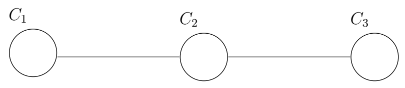


Figure 5.1: Chaining services.

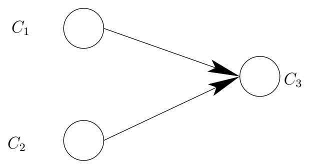


Figure 5.2: Combining selectivities.

Consider two arbitrary services C_i and C_j. If there is a precedence constraint from C_i to C_j, we need to enforce it. But if there is none, meaning that C_i and C_j are independent, we may still introduce a (fake) edge, say from C_j to C_i, in the mapping, meaning that the output of C_j is fed as input to C_i. If the selectivity of C_j is small ($\sigma_j < 1$), then it shrinks each data set, and C_i will operate on data sets of reduced volume. As a result, the cost of C_i will decrease in proportion to the volume reduction, leading to a better solution than running both services in parallel. Basically, there are two ways to decrease the final cost of a service: (i) map it on a fast server; and (ii) map it as a successor of a service with small selectivity. In general, we have to organize the execution of the application by assigning a server to each service and by deciding which service will be a predecessor of which other service (therefore building an execution graph, or plan), with the goal of minimizing the objective function. The edges of the execution graph must include all the original dependence edges of the application. We are free to add more edges if it decreases the objective function. Note that the selectivity of a service influences the execution time of all its successors, if any, in the mapping.

For instance, consider the example with three services C_1, C_2 and C_3 which are organized in a linear chain, as depicted in Figure 5.1. Then, the cost of C_2 is $\sigma_1 c_2$ and the cost of C_3 is $\sigma_1\sigma_2 c_3$. If C_i is mapped onto S_i, for $i = 1, 2, 3$, then the period is $\mathcal{P} = \max\left(\frac{c_1}{s_1}, \frac{\sigma_1 c_2}{s_2}, \frac{\sigma_1\sigma_2 c_3}{s_3}\right)$, while the latency is $\mathcal{L} = \frac{c_1}{s_1} + \frac{\sigma_1 c_2}{s_2} + \frac{\sigma_1\sigma_2 c_3}{s_3}$. Here, we also note that selectivities are independent: for instance if C_1 and C_2 are both predecessors of C_3, as in Figure 5.1 or in Figure 5.2, then the cost of C_3 becomes $\sigma_1\sigma_2 c_3$. With the mapping of Figure 5.2, the period is $\mathcal{P} = \max\left(\frac{c_1}{s_1}, \frac{c_2}{s_2}, \frac{\sigma_1\sigma_2 c_3}{s_3}\right)$, while the latency is $\mathcal{L} = \max\left(\frac{c_1}{s_1}, \frac{c_2}{s_2}\right) + \frac{\sigma_1\sigma_2 c_3}{s_3}$. We see from the latter formulas that the model neglects the cost of *joins* when combining two services as predecessors of a third one.

Formally, $\mathsf{Ancest}_j(G)$ denotes the set of all ancestors[2] of C_j in G, but only arcs from direct predecessors are kept in $\mathcal{E}$. In other words, if $(C_i, C_j) \in \mathcal{G}$, then we must have $C_i \in \mathsf{Ancest}_j(G)$ [3]. Given a plan G, the execution time of a service C_i is $cost_i(G) = \left(\prod_{C_j \in \mathsf{Ancest}_i(G)} \sigma_j\right) \times \frac{c_i}{s_{alloc(i)}}$. We note $L_G(C_i)$ the completion time of

[2]The ancestors of a service are the services preceding it, and the predecessors of their predecessors, and so on.

[3]Equivalently, $\mathcal{G}$ must be included in the transitive closure of $\mathcal{E}$.

service C_i with the plan G, which is the length of the path from an entry node to C_i, where each node is weighted with its execution time. We can now formally define the period $\mathcal{P}$ and the latency $\mathcal{L}$ of a plan G:

$$\mathcal{P}(G) = \max_{(C_i,S_u)\in\mathcal{C}} cost_i(G) \quad \text{and} \quad \mathcal{L}(G) = \max_{(C_i,S_u)\in\mathcal{C}} L_G(C_i).$$

Example with independent services. Consider a problem instance with three services C_1, C_2 and C_3 without precedence constraints. Assume that $c_1 = 1$, $c_2 = 4$, $c_3 = 10$, and that $\sigma_1 = \frac{1}{2}$, $\sigma_2 = \sigma_3 = \frac{1}{3}$. Suppose that we have three servers of respective speeds $s_1 = 1$, $s_2 = 2$ and $s_3 = 3$. What is the mapping which minimizes the period? and same question for the latency? We have to decide for an assignment of services to servers, and to build the best plan.

For the period optimization, we can look for a plan with a period smaller than or equal to 1. In order to obtain an execution time smaller than or equal to 1 for service C_3, we need the selectivity of C_1 and C_2, and either server S_2 or server S_3. Server S_2 is fast enough to render the time of C_3 smaller than 1, so we decide to assign C_3 to S_2. Service C_2 also needs the selectivity of C_1 and a server of speed strictly greater than 1 to obtain an execution time less than 1. Thus, we assign C_1 to S_1 and make it a predecessor of C_2. In turn we assign C_2 to S_3 and make it a predecessor of C_3. We obtain a period of $\min\left(\frac{1}{1}, \frac{1}{2}\frac{4}{3}, \frac{1}{2\times 3}\frac{10}{2}\right) = 1$. It is the optimal solution. In this plan, the latency is equal to $1 + \frac{4}{6} + \frac{10}{12} = \frac{5}{2}$.

For the latency optimization, we have a first bound: $\frac{5}{2}$. Because of its cost, service C_3 needs at least one predecessor. If C_1 is the only predecessor of C_3, we have to assign C_3 to S_3 in order to keep the latency under $\frac{5}{2}$. The fastest computation time that we can then obtain for C_3 is $\frac{1}{2} + \frac{1}{2}\frac{10}{3}$, with C_1 assigned to S_2. In this case, the fastest completion time for C_2 is $\frac{5}{2}$: this is achieved by letting C_2 be a successor of C_1 in parallel with C_3. Suppose now that C_2 is a predecessor of C_3, and that there is an optimal solution in which C_2 is the only predecessor of C_3. Independently of the choice of the servers assigned to C_1 and C_2, if we put C_1 without any predecessor, it will end before C_2. So, we can make it a predecessor of C_3 without increasing its completion time. So, we are looking for a solution in which C_1 and C_2 are predecessors of C_3. There are three possibilities left: (i) C_1 is a predecessor of C_2; (ii) C_2 is a predecessor of C_1; and (iii) C_1 and C_2 have no predecessors. In the first two cases, we compute for each service a cost weighted by the product of the selectivities of its predecessors. Then, we associate the fastest server to the service with the longest weighted cost and so on. We obtain $\frac{5}{2}$ in both cases. For the last case, we know that the real cost of C_1 will have no influence on the latency, hence we assign it to the slowest server S_1. The weighted cost of the remaining services is 4 for C_2 and $\frac{10}{6}$ for C_3. So, we assign S_3 to C_2 and S_2 to C_3. We obtain a latency of $\frac{4}{3} + \frac{1}{2\times 3}\frac{10}{2} = \frac{13}{6}$. We cannot obtain a strictly faster solution if C_2 is not a predecessor of C_3. As a result, $\frac{13}{6}$ is the optimal latency. In this optimal plan for the latency, the period is $\frac{4}{3}$.

This simple example with no precedence constraints illustrates the challenges arising from filtering applications, even if we do not account for communication costs. We now proceed to complexity results, first with no communication costs in Section I.2, and then with communication costs in Section I.3.

I.2 Complexity results with no communication costs

The detailed presentation of complexity results with proofs can be found in [C26]. We outline the results first for homogeneous platforms, and then in the heterogeneous case.

I.2.1 Homogeneous platforms: identical resources

The problem of minimizing the period with precedence constraints and identical resources was shown to have polynomial complexity in [114, 28]. We extended this result to the latency minimization problem, and give a complicated polynomial algorithm to solve the problem (see [C26]). In order to give an insight of the problem complexity, we present the simpler algorithm for the case with no precedence constraints, which runs in time $O(n^2)$ (see Algorithm 1).

The proof that this algorithm returns the optimal solution is already very technical. We show that Algorithm 1 verifies the following properties:

- (A) $L_G(C_1) \leq L_G(C_2) \leq \cdots \leq L_G(C_p)$;
- (B) $\forall i \leq n$, $L_G(C_i)$ is optimal.

Because the latency of any plan G' is the completion time of its last node (a node C_i such that $\forall C_j, L_{G'}(C_i) \geq L_{G'}(C_j)$), property (B) shows that $\mathcal{L}(G)$ is the optimal latency. We prove properties (A) and (B) by induction on i: for every i we prove that $L_G(C_i)$ is optimal and that $L_G(C_1) \leq L_G(C_2) \leq \cdots \leq L_G(C_i)$ (see [C26]).

The bi-criteria problem consists in finding a plan G whose period does not exceed K and whose latency is minimal, given a bound on the period K. Both mono-criterion problems are polynomial, and we derive a bi-criteria polynomial algorithm

```
Data: $n$ services with selectivities $\sigma_1, ..., \sigma_p \leq 1$ without precedence
      constraints, $\sigma_{p+1}, ..., \sigma_n > 1$, and ordered costs $c_1 \leq \cdots \leq c_p$
Result: a plan $G$ optimizing the latency
$G$ is the graph reduced to node $C_1$;
for $i = 2$ to $n$ do
    for $j = 0$ to $i - 1$ do
        Compute the completion time $L_j(C_i)$ of $C_i$ in $G$ with predecessors
        $C_1, ..., C_j$;
    end
    Choose $j$ such that $L_j(C_i) = \min_k\{L_k(C_i)\}$;
    Add the node $C_i$ and the edges $C_1 \to C_i, \ldots, C_j \to C_i$ to $G$;
end
```

Algorithm 1: Latency minimization, identical servers, no precedence constraints.

to solve this latter problem. The algorithm is based on the algorithm which minimizes the latency, and its complexity in the general case with precedence constraints is at most $O(n^6)$ (see [C26]).

I.2.2 Heterogeneous platforms

With heterogeneous resources, we show that both period and latency minimization problems are NP-hard, even for services without precedence constraints. Thus, bi-criteria problems on heterogeneous platforms are NP-hard.

The completeness for the period minimization problem is established through an involved reduction from RN3DM, a special instance of Numerical 3-Dimensional Matching that has been proved to be strongly NP-complete by Yu [131, 132]. The problem remains NP-hard even when all service selectivities are identical. Moreover, we prove that there exists no approximation algorithm with a constant factor, unless P=NP.

To prove the completeness of the latency minimization problem, we first show that the optimal solution has a particular structure. We then use this result to derive the NP-completeness of the problem. Given a plan G and a vertex $v = (C_i, S_u)$ of G, v is a leaf if it has no successor in G, and we define $d_i(G)$ as the maximum length (number of links) in a path from v to a leaf. Note that if v is a leaf, then $d_i(G) = 0$. Given a set of services and servers, the optimal latency can be obtained with a plan G such that, for any couple of nodes of G $v_1 = (C_{i_1}, S_{u_1})$ and $v_2 = (C_{i_2}, S_{u_2})$,

1. If $d_{i_1}(G) = d_{i_2}(G)$, v_1 and v_2 have the same predecessors and the same successors in G.
2. If $d_{i_1}(G) > d_{i_2}(G)$ and $\sigma_{i_2} \leq 1$, then $c_{i_1}/s_{u_1} < c_{i_2}/s_{u_2}$.
3. All nodes with a service of selectivity $\sigma_i > 1$ are leaves ($d_i(G) = 0$).

The proof of this property is already very technical, and then the completeness of the problem comes from a reduction from RN3DM: we build an instance of our problem whose solution will necessarily be a linear chain. We refer the interested reader to [C26] for all the details.

I.3 Filtering applications with communication costs

With different speed processors, all problems are NP-hard, even without any communication costs. We thus consider mapping filtering applications onto homogeneous platforms: each server has the same speed, and all servers are connected to each other by communication links of equal bandwidth. Note that we do not need to specify which service is mapped onto which server, since all servers are equivalent. Instead, we have to generate the execution graph, which is not constrained to be a linear chain as in Chapter 2. Thus, even though we restrict to one-to-one mappings, a processor may communicate with several other processors, and we need to define an operation list, which details the time-steps at which every computation and every communication takes place. As before, we assume that the schedule is cyclic, so that the execution list can be specified concisely.

We consider two commonly used communication models, as discussed in Chapter 2. The *no overlap* communication model requires that at any point, a server can either compute, or receive an incoming communication, or send an outgoing communication. This models single threaded machines where every operation is serialized. We define two variants for this model, one where we enforce in-order execution, and another where we allow out-of-order execution (which means interleaving communications and computations of different data sets) so as to reduce the idle-time incurred by the serial ordering of the communications[4]. In contrast, the *overlap* communication model considers the situation where a server can compute and send/receive communications at the same time. This calls for multi-threaded machines and parallel communications. In all models, both computations and communications are non-preemptive, which means that they cannot be interrupted once initiated. Also, communications are synchronous (by rendez-vous between the sender and the receiver). This synchronization between servers can cause idle times.

Our main findings is that computing the period or the latency in all these models turns out to be difficult. As already stated, the minimization problems (finding the optimal plan to minimize the period or the latency) are all NP-hard. This result is surprising, since polynomial algorithms exist for homogeneous machines when we do not model communication, as discussed in Section I.2. Therefore, modeling communication costs explicitly has a huge impact on the difficulty of mapping filtering services.

In addition, and quite unexpectedly, the "orchestration" problems (given an execution graph, find the optimal operation list) also are of combinatorial nature, similarly to the cases exhibited in Section 2 in the context of pipelined applications with no filtering. For the communication model with overlap, and given an execution graph, we provide a polynomial algorithm which determines the operation list that leads to the best period. However the same problem turns out to be NP-hard in the model with no overlap. The counterpart results for the latency are even more striking: this problem turns out to be NP-hard for all models (while determining the best period is polynomial for the model with overlap). All these results imply technically involved reduction proofs, that can be found in [C32].

I.4 Conclusion

In this section, we have explored the problem of mapping filtering streaming applications on large-scale platforms, and discussed communication models and their impact. The following important problems have been addressed: (i) Given an execution graph, what is the complexity of computing the period or the latency? (ii) What

[4]Note that these two variants can been introduced for linear chain applications as well. We did not address this level of detail in Chapter 2 because the variants have no impact on complexity results in this case: we only need to modify the constraints of the operation list to take them into account.

is the complexity of the general period or latency minimization problem?

We have been able to provide the complexity of all optimization problems, with or without communication costs, thereby providing solid theoretical foundations for the study of filtering streaming applications. Several of these results apply to regular pipelined applications, which broadens the scope and significance of these results to quite a large applicative framework, such as the one discussed in the previous chapters.

In the next section, we are back to classical pipelined applications, but we consider more complex dependency graphs, such as forks and general DAGs.

II Fork and DAG application patterns

While linear chain graphs occur in many applications in the domains of image processing, computer vision, query processing, etc, fork graphs are mandatory to distribute files or databases in master-worker environments. Both graphs are important but also simple enough so that the design of optimal mappings is well understood in some simple frameworks. We have already extensively discussed the case of linear chains, and we extend this discussion to fork and fork-join graphs in Section II.1. Fork graphs are more difficult to tackle, because there are more opportunities for parallelism, hence a wider combinatorial space to explore when searching for good mappings.

The study of general directed acyclic graphs (DAGs) is even more complex, and most problems are already known to be NP-hard for such applications. We briefly discuss the case of DAG applications in Section II.2.

II.1 Fork graphs

A fork graph of $n+1$ stages S_k, $0 \leq k \leq n$ is illustrated on Figure 5.3. S_0 is the root stage while S_1 to S_n are independent stages that can be executed simultaneously for a given data set. Stage S_k $(0 \leq k \leq n)$ performs a number of w_k computations on each data set. As for the linear graph, consecutive data sets are fed into the fork. Each data set first proceeds through stage S_0, which outputs its results, of size δ_0, to all the other stages. The first stage S_0 receives an input of size δ_{-1} from the outside world, while the other stages S_k, $1 \leq j \leq n$, may return their results, of size δ_k, to the outside world.

For a fork graph, it is natural to map any partition of the graph onto the processors. Assume such a partition with q sets, where $q \leq p$. The first set of the partition contains the root stage S_0 and possibly other independent stages (say S_1 to S_k without loss of generality), while the other sets only contain independent stages chosen from S_{k+1} to S_n. Assume that the first set (with the root stage) is assigned to P_1, and that the $q-1$ remaining sets are assigned to $P_2, \ldots, P_q$. Defining the period requires to make several hypotheses on the communication model:

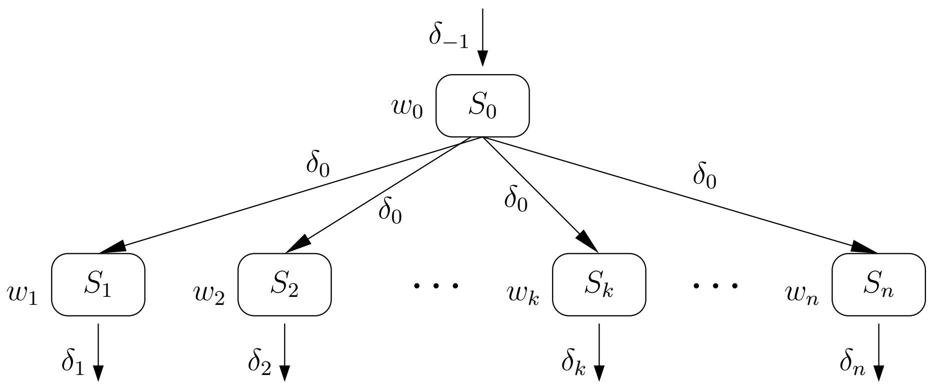


Figure 5.3: The application fork.

- A flexible model would allow P_1 to initiate the communications to the other processors immediately upon completion of the execution of S_0, while a stricter model (say, with a single execution thread) would allow the communications to start only after P_1 has completed all its computations, including those for stages S_1 to S_k.
- Either way, we need to specify the ordering of the communications. This is mandatory for the one-port model without overlap, obviously, but this is also true for the bounded multi-port model with overlap: a priori, there is no reason for the $q-1$ communications to take place in parallel; the scheduler might decide to send some messages first and others later.

Also, we want to allow replication of dealable stages, and then the situation gets even more complicated since the latency depends upon whether the model is flexible or strict, and it is hugely impacted by the communication ordering. Data-parallel stages are also considered.

We therefore restrict this study to a framework with no communication costs nor overheads, similarly to the simplified model presented in Chapter 2. In this framework, the period is dictated by the critical resource. We detail how to compute the latency for fork graphs, which requires some additional notations. Assume a partition of the $n+1$ stages into q sets $\mathcal{I}_r$, where $1 \le r \le q \le p$. Without loss of generality, the first set $\mathcal{I}_1$ contains the root stage and is assigned to the set of k processors $P_{q_1}, \dots, P_{q_k}$. Let $trav_r$ be the delay of the r-th set, $1 \le r \le q$, computed as if it was a stage interval of a pipeline graph, i.e., using the formulas of Chapter 2 to account for data-parallelism or replication. We use a flexible model where the computations of set $\mathcal{I}_r$, $r \ge 2$, can start as soon as the computation of stage S_0, from which it has an input dependence, is completed. In other words, there is no need to wait for the completion of all tasks in $\mathcal{I}_1$ to initiate the other sets $\mathcal{I}_r$, we only wait for S_0 to terminate. Let s_0 be the speed at which S_0 is processed, hence $s_0 = \sum_{u=1}^{k} s_{q_u}$ if $\mathcal{I}_1$ is data-parallelized, and $s_0 = \min_{1 \le u \le k} s_{q_u}$ if $\mathcal{I}_1$ is replicated.

We then derive the latency of the mapping as

$$\mathcal{L} = \max\left(trav_1, \frac{w_0}{s_0} + \max_{2 \le r \le q} trav_r\right).$$

We are now ready to derive complexity results. Since communication costs are neglected, we consider either homogeneous platforms with identical speed processors, or heterogeneous ones with different speed processors.

Theorem 13. *For homogeneous platforms, the optimal fork mapping which minimizes the period can be determined in polynomial time, with or without data-parallelism.*

This result actually holds for any DAGs when replication is allowed, since the optimal solution consists in replicating the whole DAG on all processors, when they are all identical.

Theorem 14. *For homogeneous platforms, the optimal homogeneous fork mapping which minimizes the latency can be determined in polynomial time, with or without data-parallelism. The problem becomes NP-hard for a heterogeneous fork.*

A homogeneous fork is such that stages S_1 to S_n have the same computational cost. Then we are able to derive a sophisticated bi-criteria dynamic programming algorithm, for both cases with or without data-parallelism, whose complexity is always bounded by $O(n^3p^3)$. See [J10] for the details. In the general case (heterogeneous fork), the problem is NP-hard, and the reduction comes from 2-PARTITION [54].

On heterogeneous platforms, only the homogeneous fork cases with no data-parallelism can be solved in polynomial case, while all remaining cases are NP-hard. With data-parallelism, even with a homogeneous fork, we can build an instance with only two fork stages, which is indeed a linear chain graph, and thus the results are identical to those of Chapter 3:

Theorem 15. *For heterogeneous platforms with data-parallelism, finding the optimal mapping for a homogeneous fork, for any objective (minimizing latency or period), is NP-complete.*

Without data-parallelism, the results are more interesting:

Theorem 16. *For heterogeneous platforms without data-parallelism, the optimal homogeneous fork mapping for any objectives can be determined in polynomial time. The problem becomes NP-hard for a heterogeneous fork.*

For the polynomial case, a sophisticated algorithm is provided in [J10], which expresses the solution with intervals of similar speed processors. We derive a polynomial dynamic programming algorithm in $O(p^5)$, which must be executed at each step of a binary search on the period and/or latency. The completeness in the heterogeneous case comes here again from 2-PARTITION [54], see [J10].

Extension to fork-join graphs. We have concentrated in this section on the complexity of mapping algorithms for fork graphs, but it is also very common to have fork-join graphs, in which a final stage, S_{n+1}, is gathering all the results and performing some final computations. We briefly explain that all the complexity results obtained for fork graphs can be extended to fork-join graphs. In other words, the complexity is not modified by the addition of the final stage. Clearly, all the problem instances which are NP-complete for a simple fork are still NP-complete for a fork-join graph. The question is to check whether we can extend the polynomial algorithms to handle fork-join or not. The answer is positive in all cases. We do not formally present these new algorithms, but rather give an insight on how to design the extensions.

First, consider the polynomial entries on homogeneous platforms. The straightforward algorithm to minimize the period for a fork is still working for a fork-join, since the replication of the whole graph on all the processors still provides the optimal period (it actually works for any DAG). Minimizing the latency or a bi-criteria algorithm was requiring a dynamic programming algorithm for a homogeneous fork (the problem being NP-hard for a heterogeneous fork). The dynamic programming algorithms used in the proof of Theorem 14 extend to fork-join graphs by adding two external loops, the first over the number of stages which belong to the same interval as the final stage S_{n+1}, and the second over the number of processors onto which these latter stages are mapped. We should also consider the case in which S_0 and S_{n+1} are in the same interval. The rest of the algorithms is unchanged. Taking the new loops into account, we add a factor $O(np)$ to the complexity, which finally becomes $O(n^4p^4)$.

The only polynomial algorithm on heterogeneous platforms is for a homogeneous fork without data-parallelism. This corresponds to the algorithm of Theorem 16, which executes a binary search, and a dynamic programming computation at each iteration of the binary search. For a homogeneous fork-join graph, we are still able to describe the form of an optimal solution, using intervals of processors with consecutive speeds. One of the processor intervals must be in charge of S_{n+1}, it can either be the one in charge of S_0 or another one. We need to distinguish both cases, and to add a loop on the first processor of the interval which handles S_{n+1} whenever it is different from the one which handles S_0. The formula are then slightly modified to take into account the time of the final computations, but the algorithm remains similar. We have added $O(p)$ to the complexity, leading to a total complexity of $O(p^6)$ for each iteration of the binary search.

On the theoretical side, we see that extending all the complexity results to fork-join graphs is not very difficult. But we believe that this extension is worth mentioning, because of the importance of fork-join graphs in many practical applications. In fact, numerous parallel applications can be expressed with the master-worker paradigm: the master initiates some computations, and then distributes (scatters) data to the workers (in our case, stages $S_1, ..., S_n$ of the fork-join). Results are then collected and combined (join operation).

We move the discussion to general DAGs in the next section.

II.2 General DAGs

As discussed before, most problems are NP-hard when tackling general DAGs. Therefore, for such applications, rather than deriving complexity results, we have designed multi-criteria heuristics aiming at proposing efficient solutions to our optimization problems.

We refer to [J11],[J15] for results on non-pipelined DAGs, i.e., DAGs which are executed only once as in classical scheduling. Here again, we targeted "dynamic" platforms subject to unrecoverable interruptions (see Chapter 4), and the aim was to minimize the makespan, or latency, while tolerating a given number of failures. Such problems have been investigated for various communication models. We recently extended this work to the case of pipelined DAGs, therefore designing heuristics to optimize the latency of streaming applications under throughput and reliability constraints (see [C35]).

In the two next sections, we target more complex applications which require more than mapping pipelined tasks onto some set of resources, as discussed so far.

III Replica placement in tree networks

This section deals with the general problem of replica placement in tree networks. Informally, there are clients issuing requests to be satisfied by servers. The clients are known (both their position in the tree and their number of requests), while the number and location of the servers are to be determined. A client is a leaf node of the tree, and its requests can be served by one or several internal nodes. Initially, there are no replica; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit. Nodes equipped with a replica, also called servers, can only serve clients located in their subtree (so that the root, if equipped with a replica, can serve any client); this restriction is usually adopted to enforce the hierarchical nature of the target application platforms, where a node has knowledge only of its parent and children in the tree.

The rule of the game is to assign replicas to nodes so that some optimization function is minimized. Typically, this optimization function is the total utilization cost of the servers. If all the nodes are identical, this reduces to minimizing the number of replicas. If the nodes are heterogeneous, it is natural to assign a cost proportional to their capacity (so that one replica on a node capable of handling 200 requests is equivalent to two replicas on nodes of capacity 100 each).

We point out that the distribution tree (clients and nodes) is fixed in our approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery [127, 74, 75, 38, 126, 89]. The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide a hierarchical and distributed access to replicas of the original data. On the contrary, in other, more decentralized, applications (e.g., allocating Web mirrors in distributed networks), a two-step approach is used [91, 125, 105, 76, 117, 80]: first determine a “good” distribution tree in an arbitrary interconnection graph, and then determine a “good” placement of replicas among the tree nodes. Both steps are interdependent, and the problem is much more complex, due to the combinatorial solution space (the number of candidate distribution trees may well be exponential).

We first detail the framework, and in particular the different policies that can be enforced for replica placement. We describe and compare access policies in Section III.1. Then we provide complexity results for different instances of the problem in Section III.2. Finally we conclude in Section III.3.

III.1 Access policies

In most papers from the literature, all requests of a client are served by the closest replica, i.e., the first replica found in the unique path from the client to the root in the distribution tree. This *Closest* policy is simple and natural, but may be unduly restrictive, leading to a waste of resources. We introduce two new policies.

In the first one, we keep the restriction that all requests from a given client are processed by the same replica, but we allow client requests to “traverse” servers so as to be processed by other replicas located higher in the path (closer to the root). We call this approach the *Upwards* policy. The trade-off to explore is the following: the *Closest* policy assigns replicas at proximity of the clients, but may need to allocate too many of them if some local subtree issues a great number of requests. The *Upwards* policy will ensure a better resource usage, load-balancing the process of requests on a larger scale; the possible drawback is that requests will be served by remote servers, likely to take longer time to process them. Taking QoS constraints into account would typically be more important for the *Upwards* policy.

In the second approach, we further relax access constraints and grant the possibility for a client to be assigned several replicas. With this *Multiple* policy, the processing of a given client’s requests will be split among several servers located in the tree path from the client to the root. Obviously, this policy is the most flexible, and likely to achieve the best resource usage. The only drawback is the (modest) additional complexity induced by the fact that requests must now be tagged with the replica server identifier in addition to the client identifier.

The comparison between access policies are done in a framework with identical node capacities, thus the problem amounts at minimizing the number of servers.

Impact of the policies on the existence of a solution. We first show the impact of the policies on a very simple instance of the problem. In this example there are two nodes, B being the unique child of A, the tree root (see Figure 5.4). Each node can process $W = 1$ request.

- If B has one client child making 1 request, the problem has a solution with all three policies, placing a replica on B or on A indifferently (Figure 5.4(a)).

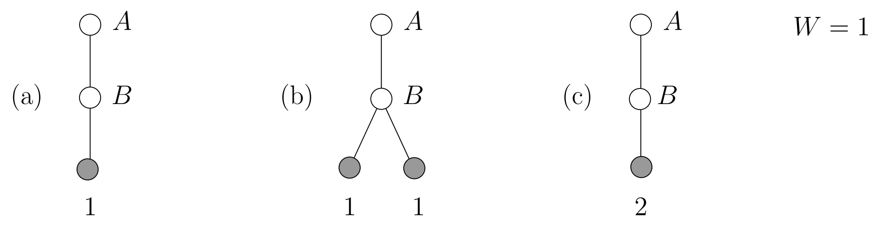


Figure 5.4: Solution existence.

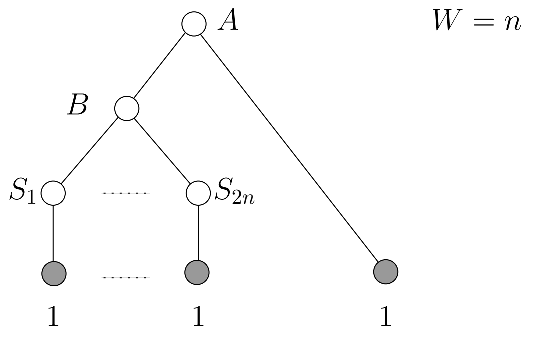


Figure 5.5: Solution cost: *Upwards* versus *Closest*.

- If B has two client children, each making 1 request, the problem has no more solution with *Closest*. However, we have a solution with both *Upwards* and *Multiple* if we place replicas on both nodes. Each server will process the request of one of the clients (Figure 5.4(b)).

- Finally, if B has only one client child making 2 requests, only *Multiple* has a solution since we need to process one request on B and the other on A, thus requesting multiple servers (Figure 5.4(c)).

Impact of the policies on the cost of a solution. We construct an instance of the problem where the *Upwards* policy is arbitrarily better than the *Closest* policy. We consider the tree network of Figure 5.5, where there are $2n + 2$ internal nodes of capacity $W = n$, and $2n + 1$ clients, each of them making one request. With the *Upwards* policy, we place three replicas in A, B and S_{2n}. All requests can be satisfied with these three replicas. When considering the *Closest* policy, first we need to place a replica in A to cover its client. Then,

- Either we place a replica on B. In this case, this replica is handling n requests, but there remain n other requests from the $2n$ clients in its subtree that cannot be processed by B. Thus, we need to add n replicas among $S_1, \dots, S_{2n}$.
- Otherwise, $n-1$ requests of the $2n$ clients in the subtree of B can be processed by A in addition to its own client. We need to add $n+1$ extra replicas among $S_1, \dots, S_{2n}$.

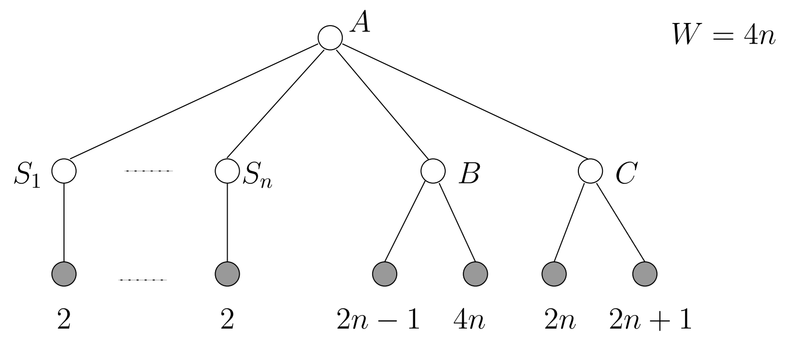


Figure 5.6: Solution cost: *Multiple* versus *Upwards*.

In both cases, we are placing $n+2$ replicas, instead of the 3 replicas needed with the *Upwards* policy, hence a performance factor of $\frac{n+2}{3}$. This proves that *Upwards* can be arbitrary better than *Closest*, even in the simple case with homogeneous servers.

The second comparison is between *Multiple* and *Upwards*. We build an instance of the replica placement problem where both access policies have a solution, but the solution of *Multiple* is arbitrarily better than the solution of *Upwards*. Consider the instance represented in Figure 5.6, with $3+n$ nodes of capacity $W = 4n$. The root A has $n+2$ children nodes B, C and $S_1, ..., S_n$. Node B has two client children, one with $2n-1$ requests and the other with $4n$ requests. Node C has two client children, one with $2n$ requests and the other with $2n+1$ requests. Each node S_i $(1 \leq i \leq n)$ has a unique child, a client with 2 requests.

- The *Multiple* policy assigns 3 replicas to A, B and C. B handles the $4n$ requests of its second client, while the other client is served by A. C handles $2n$ requests from both of its clients, and the 1 remaining request is processed by A. Server A therefore processes $(2n-1)+1 = 2n$ requests coming up from B and C. Requests coming from the n remaining nodes sum up to $2n$, thus A is able to process all of them.
- For the *Upwards* policy, we need to assign replicas everywhere. Indeed, with this policy, C cannot handle more than $2n+1$ requests since it is unable to process requests from both of its children, and thus A has $(2n-1)+2n$ requests coming from B and C. It cannot handle any of the $2n$ remaining requests, and thus each remaining node S_i $(1 \leq i \leq n)$ must process requests coming from its own client. This leads to a total of $n+3$ replicas.

The performance factor is thus $\frac{n+3}{3}$, which can be arbitrarily big when n becomes large. This proves that *Multiple* can be arbitrary better than *Upwards*, even in the simple case with homogeneous servers.

III.2 Complexity results

One major goal of this study is to assess the impact of the access policy on the problem with homogeneous versus heterogeneous servers, and without QoS versus with QoS constraints, as explained below.

We consider a tree $\mathcal{T} = \mathcal{C} \cup \mathcal{N}$, where the clients $\mathcal{C}$ are leaves of the tree. Each client $i \in \mathcal{C}$ has r_i requests; each node $j \in \mathcal{N}$ has processing capacity W_j and storage cost $sc_j = W_j$. We need to decide which node is equipped with a replica, and thus becomes a server ($j \in \mathcal{R}$, where $\mathcal{R}$ is the set of replicas). This problem comes in two flavors, either with homogeneous nodes ($W_j = W$ for all $j \in \mathcal{N}$) (Replica Counting, the goal is to minimize the number of servers), or with heterogeneous nodes, i.e., servers with different capacities/costs (Replica Cost, the goal is to minimize the total storage cost $\sum_{j \in \mathcal{R}} sc_j$). The comparison between policies was done in the homogeneous case, and thus holds true for the heterogeneous case.

In some problem instances, we add QoS constraints to clients, in terms of distance (in number of hops) between a client and its server(s). For each client, a maximum distance is fixed, which should not be exceeded.

In the single server version of the problem, we need to find a server $server(i)$ for each client $i \in \mathcal{C}$. $\mathcal{R}$ is the set of replica, i.e., the servers chosen among the nodes in $\mathcal{N}$. The constraint is that server capacities cannot be exceeded: this translates into

$$\sum_{i \in \mathcal{C}, server(i)=j} r_i \leq W_j \quad \text{for all } j \in \mathcal{N}.$$

The objective is to find a valid solution of minimal storage cost $\sum_{j \in \mathcal{R}} W_j$. As outlined before, there are two variants of the single server version of the problem, namely the *Closest* and the *Upwards* strategies.

In the *Multiple* policy with multiple servers per client, for any client $i \in \mathcal{C}$ and any node $j \in \mathcal{N}$, $r_{i,j}$ is the number of requests from i that are processed by j ($r_{i,j} = 0$ if $j \notin \mathcal{R}$, and $\sum_{j \in \mathcal{N}} r_{i,j} = r_i$ for all $i \in \mathcal{C}$). The capacity constraint now writes

$$\sum_{i \in \mathcal{C}} r_{i,j} \leq W_j \quad \text{for all } j \in \mathcal{R},$$

while the objective function is the same as for the single server version.

The decision problems associated with the previous optimization problems are easy to formulate: given a bound on the number of servers (homogeneous version) or on the total storage cost (heterogeneous version), is there a valid solution that meets the bound?

Table 5.1 captures the complexity results. These complexity results are all new, except for the *Closest*/Homogeneous combination, and proofs can be found in [J8].

The NP-completeness of the *Upwards*/Homogeneous case comes as a surprise, since all previously known instances were shown to be polynomial, using dynamic programming algorithms. In particular, the *Closest*/Homogeneous variant remains polynomial when adding communication costs [38] or QoS constraints [89]. We provide an

	REPLICA COUNTING		REPLICA COST
	Homogeneous		**Heterogeneous**
	No QoS	**With QoS**	**No/With QoS**
Closest	polynomial [38, 89]	polynomial [89]	NP-complete
Upwards	NP-complete	NP-complete	NP-complete
Multiple	polynomial	NP-complete	NP-complete

Table 5.1: Complexity results

elegant algorithm to show the polynomial complexity of the *Multiple*/Homogeneous problem. This multi-pass algorithm is very involved and can be found in [J8]. Another important contribution is the NP-completeness of the *Multiple* policy with QoS constraints for the homogeneous case, which gives a clear insight on the additional complexity introduced by QoS constraints. Also, all problems become NP-complete when dealing with resource heterogeneity (REPLICA COST problem).

Note that previous NP-completeness results involved general graphs rather than trees, and the combinatorial nature of the problem came from the difficulty to extract a good replica tree out of an arbitrary communication graph [91, 105]. Here the tree is fixed, but the problem remains combinatorial due to QoS or resource heterogeneity.

We also provide an expression of the REPLICA PLACEMENT optimization problem in terms of an integer linear program. We deal with the most general instance of the problem on a heterogeneous tree, including QoS constraints, and bounds on server capacities. We derive a formulation for each of the three server access policies, namely *Closest*, *Upwards*, and *Multiple*. This is an important extension to a previous formulation due to [77].

The linear program, which can be found in [J8], contains boolean or integer variables, because it does not make sense to assign half a request or to place one third of a replica on a node. Thus, it must be solved in integer values if we wish to obtain an exact solution to an instance of the problem, and there is no efficient algorithm to solve integer linear programs (unless P=NP). For each access policy, there is a large number of variables, and the problem cannot be solved for platforms of size $s > 50$, where $s = |\mathcal{N}| + |\mathcal{C}|$. Thus we cannot use this approach for large-scale problems.

However, this formulation is extremely useful as it leads to an absolute lower bound: we can solve the integer linear program over the rationals. In this case, all constraints are relaxed and we assume that all variables can take rational values. The optimal solution of the relaxed program can be obtained in polynomial time (in theory using the ellipsoid method [109], in practice using standard software packages [30, 57]), and the value of its objective function provides an absolute lower bound on the cost of any valid (integer) solution. For all practical values of the problem size, the rational linear program returns a solution in a few minutes. We tested up to several thousands of nodes and clients, and we always found a solution within ten seconds. Of course the relaxation makes the most sense for the *Multiple* policy, because several fractions of servers are assigned by the rational program.

Moreover, for the *Multiple* policy, we prove that the solution of the linear program, when solved with all variables being rational except a small subset of them, is an achievable bound for the *Multiple* problem, and we can build an exact solution in polynomial time, based on the LP solution. This interesting result is detailed in [J8].

III.3 Conclusion

In this section, we have introduced and analyzed two important new policies for the replica placement problem. The *Upwards* and *Multiple* policies are natural variants of the standard *Closest* approach, and it may seem surprising that they have not already been considered in the published literature.

On the theoretical side, we have fully assessed the complexity of the *Closest*, *Upwards*, and *Multiple* policies, both for homogeneous and heterogeneous platforms, and with or without QoS constraints. The polynomial complexity of the *Multiple* policy in the homogeneous case without QoS constraints is quite unexpected, and we have provided an elegant algorithm to compute the optimal cost for this policy. When adding QoS constraints, the same problem becomes NP-complete, which illustrates the additional complexity induced by such constraints. Not surprisingly, all three policies turn out to be NP-complete for heterogeneous nodes, which provides yet another example of the additional difficulties induced by resource heterogeneity.

On the practical side, we have designed in [J8] several heuristics for the *Closest*, *Upwards*, and *Multiple* policies, and we have compared their performance for several problem instances with or without QoS constraints. In the experiments, the total cost was the sum of the server capacities (or their number in the homogeneous case). The impact of the new policies is impressive: the number of trees which admit a solution is much higher with the *Upwards* and *Multiple* policies than with the *Closest* policy. Finally, we point out that the absolute performance of the heuristics is quite good, since their cost is close to the optimal solution based upon the solution of the integer linear program.

In recent work [C33], we have provided an algorithm to build a single server solution, which is guaranteed to use no more than two times more servers than the optimal *Multiple* solution, given some constraints on the problem instance. This is a very interesting result, given that the *Upwards* problem on homogeneous platforms is NP-hard, and that some applications may not support multiple allocations. Even though the constraints on the trees are quite restrictive, the procedure can be applied on any tree and still return good *Upwards* solutions, even if the application tree does not allow for a guarantee on the solution. We expect that the ratio of 2 should be achievable in most practical situations.

In the following application, operators need to download basic objects from servers, thus it amounts to a replica placement problem in a more general setting than a tree network. This replica placement problem is mixed with a mapping problem, since operators also need to be mapped onto processors in order to perform some computation. Through this last case study, we therefore mix all difficulties encountered so far.

IV In-network stream processing

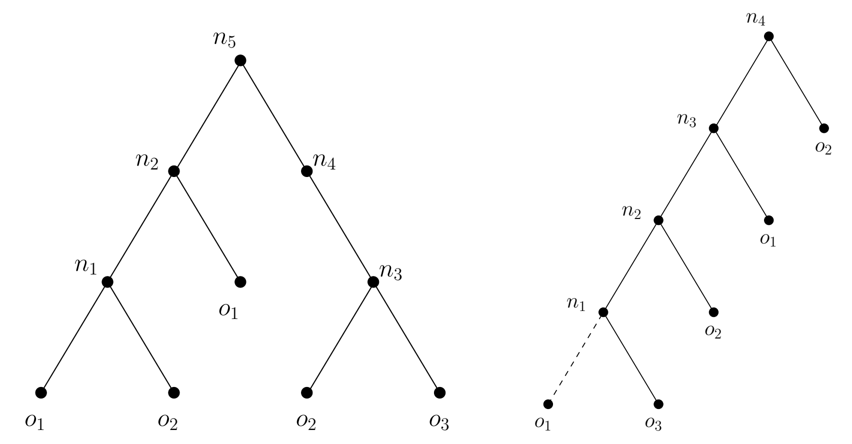
We consider in this section the execution of applications structured as trees of operators, where the leaves of the tree correspond to basic data objects that are distributed over servers in a distributed network. Each internal node in the tree denotes the aggregation and combination of the data from its children, which in turn generates new data that is used by the node's parent. The computation is complete when all operators have been applied up to the root node, thereby producing a final result. We consider the scenario in which the basic data objects are constantly being updated, meaning that the tree of operators must be applied continuously. The goal is to produce final results at some desired rate. This problem is called *stream processing* [13] and arises in several domains.

In Section IV.1, we informally describe the problem and we illustrate its application fields through examples. Then we explore the problem complexity in Section IV.2. Finally we conclude in Section IV.3.

IV.1 Problem description

An important domain of application is the acquisition and refinement of data from a set of sensors [113, 93, 26]. For instance, [113] outlines a video surveillance application in which the sensors are cameras located at different locations over a geographical area. The goal of the application could be to identify monitored areas in which there is significant motion between frames, particular lighting conditions, and correlations between the monitored areas. This can be achieved by applying several operators (e.g., filters, pattern recognition) to the raw images, which are produced/updated periodically. Another example arises in the area of network monitoring [43, 122, 42]. In this case routers produce streams of data pertaining to forwarded packets. More generally, stream processing can be seen as the execution of one of more "continuous queries" in the relational database sense of the term (e.g., a tree of join and select operators). A continuous query is applied continuously, i.e., at a reasonably fast rate, and returns results based on recent data generated by the data streams. Many authors have studied the execution of continuous queries on data streams [10, 87, 33, 103, 84].

In practice, the execution of the operators must be distributed over the network. In some cases the servers that produce the basic objects may not have the computational capability to apply all operators. Besides, objects must be combined across devices, thus requiring network communication. Although a simple solution is to send all basic objects to a central compute server, it often proves unscalable due to network bottlenecks. Also, this central server may not be able to meet the desired target rate for producing results due to the sheer amount of computation involved. The alternative is then to distribute the execution by mapping each node in the operator tree to one or more servers in the network, including servers that produce and update basic objects and/or servers that are only used for applying operators. One then talks of *in-network stream processing* [113, 101, 4]. Several in-network stream



(a) Standard tree. (b) Left-deep tree.

Figure 5.7: Examples of applications structured as a binary tree of operators.

processing systems have been developed [1, 35, 68, 34, 97, 122, 32, 90]. These systems all face the same question: where should operators be mapped in the network?

In our problem, we enforce the constraint that the rate at which final results are produced, or *throughput*, is above a given threshold. This corresponds to a Quality of Service (QoS) requirement, which is almost always desirable in practice (e.g., up-to-date results of continuous queries must be available at a given frequency). Basic objects may be replicated at multiple locations, i.e., available and updated at these locations. We also discuss the case of multiple concurrent applications that compete for the servers. In this case each application has its own QoS requirement and the constraint is to respect all of them. In such a framework, a clear opportunity for higher performance with reduced resource consumption is to reuse common sub-expression between operator trees when applications share basic objects [100].

We consider two scenarios for the computing platform. The classical "non-constructive" scenario is the one that we have used so far: we are given a set of compute and network resources, and we aim at mapping the application onto the resources, while respecting QoS constraints. An objective must then be optimized, such as the number and performance of the resources used by the application. Target platforms can have different level of heterogeneity, as discussed in Chapter 2. One originality of this work is that we also consider a "constructive" scenario: either the user can build the platform from scratch using off-the-shelf components, or computing and network units are rented by a cloud provider (e.g., [5]). The goal is then to construct a distributed network dedicated to an application, which minimizes the monetary cost of the platform while ensuring that the desired throughput is achieved.

We restrict our study to trees of operators that are general binary trees (see Figure 5.7(a)) and we discuss relevant special cases (e.g., left-deep trees [69], see Figure 5.7(b)).

IV.2 Problem complexity

Unsurprisingly, most operator mapping problems are NP-hard, because downloading objects with different rates on two identical servers is the same problem as 2-PARTITION [54]. Let us consider the simplest problem class, i.e., mapping a fully homogeneous left-deep tree application [69] (see Figure 5.7(b)) without communication costs, with objects placed on a fully homogeneous set of servers, onto a fully homogeneous set of processors. The objective function consists in minimizing the number of used (or rented, in the constructive setting) processors. It turns out that even this problem is NP-hard, due to the combinatorial space induced by the mapping of basic objects that are shared by several operators. The completeness comes from 3-PARTITION [54], which is NP-complete in the strong sense. The proof can be found in [C28].

Note that this problem becomes polynomial if one adds the additional restriction that no basic object is used by more than one operator in the tree. In this case, one can simply assign operators to $\lceil n \times w/s \rceil$ arbitrary processors in a round-robin fashion, where n is the number of operators, w their computation cost and s the processor speed.

We also provide a formulation of the optimization problem as an integer linear program, see [C28] for the constructive scenario with a single dedicated application, and [C37] for the multi-application concurrent mapping on an existing platform.

IV.3 Conclusion

In this section, we have investigated the operator mapping problem of in-network stream processing applications onto a collection of heterogeneous processors. These stream processing applications come as a set of operator trees, that have to continuously download basic objects at different sites of the network and at the same time have to process this data to produce some final result. Unsurprisingly, such complex applications lead to NP-hard problems. However, we were able to formalize them as integer linear programs, which was a difficult task given the problem complexity.

In order to tackle these difficult problems, we have proposed in [C28] and [C37] several polynomial heuristics and we evaluated them via extensive simulations. For single applications we assessed the absolute performance of our heuristics with respect to the optimal solution of the linear program for homogeneous platforms and small problem instances. Our best heuristic almost always produces optimal results and outperforms the other heuristics. In the case of multiple concurrent applications, experiments demonstrated the importance of node reuse across applications. Reusing nodes leads to an important number of additional solutions, and also the quality of the solutions improves considerably. We concluded that top-down traversals of the application trees is more efficient than bottom-up approaches, and in particular the combination of a top-down traversal with a breadth-first search achieved good results.

V Summary and conclusion

This chapter has investigated applications that go beyond simple linear chains. Problems were already difficult with pipelined linear chain applications, and a more complex application setting adds yet another level of difficulty.

In Section I, we have added a selectivity parameter to each application stage, and most problems then become NP-hard, even when there are no dependence constraints between stages. The problems are very close to the traditional pipelined application framework, and once again we were able to exhibit cases for which, given a mapping (for this framework, an execution plan), it is difficult to compute the operation list which minimizes the period or the latency (complicated polynomial algorithm, or worse, NP-hardness of the problem). Even though we restricted this study to static frameworks (unlike in Chapter 4), almost all problems turn out to be NP-hard, and the polynomial instances can be solved through involved algorithms. This study helped us better understand the challenges raised in Chapter 2, and it is strongly correlated to the study of pipelined applications without filtering.

Back to standard pipelined application, we investigated more complex dependence graphs than a linear chain in Section II. For fork and fork-join graphs, we were able to extend some of the results of Chapter 3 in a simple framework with no communication costs and no overhead. However when targeting fully general DAGs, all problems became too complicated and we designed multi-criteria heuristics to offer efficient solutions to these difficult problems, adding the extra challenge of a dynamic platform subject to unrecoverable interruptions.

Section III targeted another class of problems, which are not scheduling problems but rather replica placement problems. The objective function is then to minimize the number, or the cost, of replicas, while satisfying all client requests. Curiously, only a very constrained access policy was discussed in the literature, while we demonstrated that relaxing its constraints allowed us to derive very efficient placement policies. We derived some sophisticated polynomial algorithms for the easiest problem instances, while several heuristics allowed us to illustrate the usefulness of the new policies.

To conclude the chapter, we have exhibited an even more complex application setting in Section IV, which mixes difficulties of the scheduling of pipelined applications and of the multi-criteria placement of replica. Indeed, operators need to download basic objects from a set of servers, and deciding where to download an object amounts to deciding where to place the replica for the operator which needs the object. Moreover, operators must be mapped onto a set of resources, which is a typical scheduling problem. Since data must be processed continuously at a given throughput, we are still in the framework of pipelined applications. This last challenging application turned out to be very difficult, and even the most simple instances of the problem were proved to be NP-hard. It illustrates well the combined difficulties of all applications.

As a conclusion, we recall that the previous chapters had illustrated the inherent difficulty of scheduling simple application patterns onto heterogeneous platforms. Chapter 4 showed the additional complexity which arises from dynamic platforms. This chapter has demonstrated that targeting more complex applications is even harder. Still, we were able to derive some interesting complexity results, and for the NP-hard instances, we designed several efficient polynomial heuristics.

CHAPTER 6

Conclusion and perspectives

In this chapter, we first give in Section I an overall conclusion on the work presented in this document: we summarize the challenges that we faced and the solutions that we proposed. Then we detail more precisely our current research activities, and future research directions in Section II.

I Conclusion

A large part of this document has been devoted to the study of the scheduling of simple application patterns in general, and linear chain pipelined applications in particular. Chapter 2 was devoted to the detailed presentation of this framework, and raised unexpected challenges: not only it is difficult to find the best mapping for an application given some objective function, but also it can turn out to be difficult simply to compute this objective function, given the mapping! Thus, one must enrich the mapping definition with an operation list which accurately describes at what time each operation occurs. This was a striking result since several current researches target such streaming applications, and many of them did not realize that the throughput of the application may not be dictated by a critical resource. We were able to identify such cases, and to point out which rules render the problem more difficult. For instance, replication for performance using a round-robin distribution to a set of processor, or general mapping rules, are adding one level of difficulty.

Chapter 3 gave an overview of complexity results for linear chain applications. First we exhaustively solved the mono-criterion problems, with some very involved polynomial algorithms and NP-completeness proofs. Then we demonstrated the challenges raised by a bi-criteria approach, not even to mention multi-criteria problems. Also, we restricted the study to three criteria, which are among the most important ones for a typical user: the application throughput (i.e., period minimization), the response time (i.e., latency minimization), and the reliability (i.e., failure probability minimization).

But how should we model the reliability of an application? Where does failure come from? How can we express the fact that the application is running on a dynamic platform, subject to variability and failures? Chapter 4 aimed at giving elements of answer by describing typical dynamic platforms, and proposing performance models to handle uncertainties. However, even though markovian-based models allow to express some variability, the behavior cannot accurately capture the inherent parallelism of the application. We thus investigated a non-markovian model based on timed Petri nets, but that we were only able to study in a fully determinism case so far. Concerning failures, the problems turn out incredibly difficult as soon as the failure probability is related to time. This chapter raised many interesting questions, and there surely remains a lot of work in this field (as we discuss in Section II).

Finally, we decided to abandon the simple application patterns of linear chains and to investigate more complex applications, in order to determine if the lessons learnt from the first study enable us to derive interesting results in more realistic cases. Therefore, in Chapter 5, we first slightly modified the linear chain streaming applications. For a Web service application, we added selectivities to application stages, and ended up facing similar challenges than for linear chain applications: for instance it is sometimes NP-hard to determine the operation list which minimizes some criteria, given a mapping and an execution graph. Still we were able to establish an exhaustive list of problem complexity. Back to a setting with no selectivity, one could envision slightly more complex application graphs than a linear chain. We exhibited the additional challenges for a simple fork or fork-join graph, and briefly discussed the even more complex case of general DAGs. On the borderline of scheduling problems, a replica placement optimization problem was described and new algorithmic solutions were proposed. Building upon our algorithmic knowledge, we were able to design efficient access policies and to demonstrate their importance. Finally, we concluded with an involved application of in-network stream processing which mixed all difficulties encountered so far. For such an application, since even the simpler problem instances are NP-hard, we proposed an integer-based linear program and a set of polynomial heuristics.

II Current and future working directions

Many challenges have been raised through the study presented in this document, and we detail in this section our current research interests. Also, we explain our future working directions.

Experiments on linear chain applications. The PhD thesis of Veronika Rehn-Sonigo, my first PhD student, is coming to an end (defense planned for July 7, 2009). During the next few months, we plan to design a set of multi-criteria heuristics for fully heterogeneous platforms, mixing the knowledge acquired so far for period, latency and failure probability minimization (see Chapters 2 and 3). Such a problem appears to be very challenging: since all link bandwidths are different, it seems hard to predict communication times as long as the mapping is not fully constructed. Thus, it is not easy to determine a strategy capable of simultaneously load balance computations while keeping communications under a prescribed threshold.

In collaboration with Harald Kosch in Passau University, we also plan to evaluate the performance of these heuristics through experiments, based on a pipelined version of the MPEG-4 encoder. A student in Passau is currently working on this application.

New research directions for linear chains. Together with Yves Robert, we have two master students who are currently working on linear chain applications.

Loïc Magnan is investigating the complexity of period and latency minimization, once a mapping is given, under various realistic communication models. Some of these recent results are given in Chapter 2. He is actually doing part of his master at MIT with Kunal Agrawal, in order to continue this fruitful collaboration.

With Paul Renaud-Goud, we have extended all mono-criterion results in a new framework with multiple concurrent linear chain applications. We identified cases in which the problem becomes NP-hard because of the concurrent applications, and derived some sophisticated polynomial algorithms for other cases. We are currently investigating the introduction of a new criterion, the energy minimization, and we plan to design efficient heuristics which aim at minimizing the energy, given a threshold on the throughput, under the bounded multi-port model with overlap.

Finally, we started investigating trade-offs between replication for reliability and replication for throughput maximization (deal replication) in a joint work with Loris Marchal, Yves Robert and Oliver Sinnen.

New applications. My second PhD student, Fanny Dufossé, has been working on filtering applications, together with Yves Robert and Kunal Agrawal. We are currently extending our application field and we study optimization problems which arise from network applications, such as bandwidth sharing algorithms, in collaboration with Qishi Wu (University of Memphis, USA).

Also, in collaboration with Alexandru Dobrila, Jean-Marc Nicod and Laurent Philippe (LIFC Besançon), we investigate applications for micro-factories, in which the tasks are subject to failures (contrarily to the classical approach with platform failures). Therefore, we need to introduce a failure model for tasks, which is closely related to the problem of designing a good failure model for dynamic platforms, as discussed below.

Dynamic platforms and variability. As already pointed out in this document, most challenges come from dynamic platforms, and many problems were left open in Chapter 4. We are actively working on this exciting topic, supported thanks to two research projects, StochaGrid and ALEAE (see Appendix B).

With Matthieu Gallet, Bruno Gaujal and Yves Robert, we are currently working on adding non-determinism in the timed Petri nets which we introduced in this document. This is only a first step in the study of the impact of variability onto scheduling algorithms, since this approach once again assumes that mappings are known beforehand, similarly to the PEPA model designed in Edinburgh. There is still a long way to go before we will be able to provide a tool which returns a schedule which accounts for variability.

The work conducted with Yves Robert, Arny Rosenberg and Frédéric Vivien, on divisible load applications with a sophisticated failure model, still needs to be extended to heterogeneous platforms. This is a real challenge given the complexity of the study in the homogeneous framework.

Finally, we recently initiated new discussions with Alain Girault (INRIA researcher in Grenoble) about failures, and there is certainly a lot of research waiting for us before we can come up with a good and realistic model for platform failure and variability.

APPENDIX

A

References and publication list

I References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, Jan. 2005.

[2] J. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[3] A. Agnetis, P. Detti, M. Pranzo, and M. S. Sodhi. Sequencing unreliable jobs on parallel machines. *Journal on Scheduling*, 2008. Available on-line at `http://www.springerlink.com/content/c571u1221560j432`.

[4] Y. Ahmad and U. Cetintemel. Network aware query processing for stream-based applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 456–467, 2004.

[5] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[6] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer Verlag, 1999.

[7] B. Awerbuch, Y. Azar, A. Fiat, and F. Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time. In *28th ACM Symp. on Theory of Computing*, pages 519–530. ACM Press, 1996.

[8] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Power-aware scheduling for periodic real-time systems. *IEEE Trans. Computers*, 53(5):584–600, 2004.

[9] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD'04: Proceedings of the 2004 ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418. ACM Press, 2004.

[10] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[11] F. Baccelli, G. Cohen, and B. Gaujal. Recursive equations and basic properties of timed Petri nets. Technical Report 1432, INRIA, May 1991.

[12] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity.* Wiley, 1992.

[13] B. Badcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Intl. Conf. on Very Large Data Bases*, pages 456–467, 2004.

[14] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*, pages 460–467. IEEE Computer Society Press, 1998.

[15] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *HCW'99, the 8th Heterogeneous Computing Workshop*, pages 125–133. IEEE Computer Society Press, 1999.

[16] O. Beaumont, V. Boudet, and Y. Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *HCW'2002, the 11th Heterogeneous Computing Workshop.* IEEE Computer Society Press, 2002.

[17] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS'2002).* IEEE Computer Society Press, 2002.

[18] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In *HeteroPar'2004: International Conference on Heterogeneous Computing, jointly published with ISPDC'2004: International Symposium on Parallel and Distributed Computing*, pages 296–302. IEEE Computer Society Press, 2004.

[19] M. A. Bender, S. 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.

[20] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002.

[21] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems.* Wiley-IEEE Computer Society Press, 1996.

[22] P. Bhat, C. Raghavendra, and V. 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.

[23] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[24] S. Bhatt, F. Chung, F. Leighton, and A. Rosenberg. On optimal strategies for cycle-stealing in networks of workstations. *IEEE Trans. Computers*, 46(5):545–557, 1997.

[25] S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Computers*, 37(1):48–57, 1988.

[26] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Conference on Mobile Data Management*, 2001.

[27] P. Brucker. *Scheduling Algorithms.* Springer-Verlag Berlin Heidelberg, 2004.

[28] J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Research Report 2005-40, Stanford University, Nov. 2005.

[29] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. HiHCoHP: Toward a realistic communication model for hierarchical HyperClusters of heterogeneous processors. In *International Parallel and Distributed Processing Symposium IPDPS'2001.* IEEE Computer Society Press, 2001.

[30] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *Maple Reference Manual*, 1988.

[31] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Systems*, 24(2):177–228, 1999.

[32] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the SIGMOD Intl. Conf. on Management of Data*, pages 379–390, 2000.

[33] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of ICDE*, 2002.

[34] L. Chen, K. Reddy, and G. Agrawal. GATES: A Grid-Based Middleware for Processing Distributed Data Streams. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, pages 192–201, 2004.

[35] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, Jan. 2003.

[36] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN: Graphical editor and analyzer for timed and stochastic petri nets. *Performance Evaluation*, 24(1-2):47–68, 1995.

[37] G. Ciardo and R. Zijal. Well-Defined Stochastic Petri Nets. In *Proc. 4th Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, pages 278–284. San Jose, CA, USA, 1996.

[38] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[39] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press & Pitman, 1989. `http://homepages.inf.ed.ac.uk/mic/Pubs/pubs.html`.

[40] M. Cole. *eSkel: The* **e***dinburgh* **Ske***leton* **l***ibrary. Tutorial Introduction.* Internal Paper, School of Informatics, University of Edinburgh, UK, `http://homepages.inf.ed.ac.uk/mic/eSkel`, 2002.

[41] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[42] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[43] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high-performance network monitoring with an SQL interface. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 623–633, 2002.

[44] P. Crescenzi and V. Kann. A compendium of NP optimization problems. World Wide Web document, URL: `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`.

[45] DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. `http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm`.

[46] A. Duarte, D. Rexachs, and E. Luque. A distributed scheme for fault-tolerance in large clusters of workstations. In *NIC Series, Vol. 33*, pages 473–480. John von Neumann Institute for Computing, Julich, 2006.

[47] F. Berman and G. Fox and A.J.G. Hey, editor. *Grid Computing: Making the Global Infrastructure a Reality.* Wiley & Sons, 2003.

[48] E. Fabiani and D. Lavenier. Placement of linear arrays. In *FPL 2000, 10th International Conference on Field Programmable Logic and Applications.* IEEE Computer Society Press, 2000.

[49] D. Florescu, A. Grunhagen, and D. Kossmann. Xl: A platform for web services. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research*, 2003. On-line proceedings at `http://www-db.cs.wisc.edu/cidr/program/p8.pdf`.

[50] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1998.

[51] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizatio ns. *Int. J. of High Performance Computing Applications*, 15(3):200–222, 2001.

[52] A. H. Frey and G. Fox. Problems and approaches for a teraflop processor. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 21–25. ACM Press, 1988.

[53] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12):1753–1772, 2002.

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

[55] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors. `http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf`, 2002.

[56] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in LNCS, pages 353–368, Vienna, May 1994. Springer-Verlag.

[57] GLPK: GNU Linear Programming Kit. `http://www.gnu.org/software/glpk`.

[58] N. Haenel. *User Guide for the Java Edition of the PEPA Workbench.* Internal Paper, School of Informatics, University of Edinburgh, 2003. `http://www.dcs.ed.ac.uk/pepa`.

[59] P. Hansen and K.-W. Lih. Improved algorithms for partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Computers*, 41(6):769–771, 1992.

[60] J. M. Hellerstein. Predicate migration: Optimizing queries with expensive predicates. In *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, 1993.

[61] H. Hillion and J.-M. Proth. Performance evaluation of job shop systems using timed event graphs. *IEEE Transaction on Automatic Control*, 34(1):3–9, 1989.

[62] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[63] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[64] B. Hong and V. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'2003)*. IEEE Computer Society Press, 2003.

[65] B. Hong and V. 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.

[66] B. Hong and V. K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE Trans. Parallel Distributed Systems*, 18(10):1420–1435, 2007.

[67] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[68] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003.

[69] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[70] M. A. Iqbal. Approximate algorithms for partitioning problems. *Int. J. Parallel Programming*, 20(5):341–361, 1991.

[71] M. A. Iqbal and S. H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Trans. Parallel and Distrbuted Systems*, 6(2):170–175, 1995.

[72] A. Jean-Marie. ERS: a tool set for performance evaluation of discrete event systems. `http://www-sop.inria.fr/mistral/soft/ers.html`.

[73] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, 1989.

[74] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel and Distributed Systems*, 12(6):628–637, 2001.

[75] K. Kalpakis, K. Dasgupta, and O. Wolfson. Steiner-Optimal Data Replication in Tree Networks with Storage Costs. In *IDEAS '01: Proceedings of the 2001 International Symposium on Database Engineering & Applications*, pages 285–293. IEEE Computer Society Press, 2001.

[76] M. Karlsson and C. Karamanolis. Choosing Replica Placement Heuristics for Wide-Area Systems. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[77] M. Karlsson, C. Karamanolis, and M. Mahalingam. A framework for evaluating replica placement algorithms. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA, 2002.

[78] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *J.Parallel and Distributed Computing*, 63(5):551–563, 2003.

[79] J. Kemeny and J. Snell. *Finite Markov Chains*. Springer New York, Heidelberg, Berlin, 1976.

[80] S. U. Khan and I. Ahmad. RAMM: a game theoretical replica allocation and management mechanism. In *Proc. Int. Symp. on Parallel Architectures, Algorithms and Networks ISPAN'05*. IEEE Computer Society Press, 2005.

[81] S. Khuller and Y. Kim. On broadcasting in heterogenous networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.

[82] D. Kondo, H. Casanova, E. Wing, and F. Berman. Models and scheduling mechanisms for global computing applications. In *Intl. Parallel and Distr. Processing Symp.*, 2002.

[83] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. SETI@home-massively distributed computing for SETI. *Computing in Science & Engineering*, 3(1):78–83, 2001.

[84] J. Kräme and B. Seeger. A Temporal Foundation for Continuous Queries over Data streams. In *Proceedings of the Intl. Conf. on Management of Data*, pages 70–82, 2005.

[85] D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM J. Computing*, 21:111–139, 1992.

[86] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[87] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[88] P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.

[89] P. Liu, Y.-F. Lin, and J.-J. Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems (ICPADS).* IEEE Computer Society Press, 2006.

[90] D. Logothetis and K. Yocum. Wide-Scale Data Stream Management. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[91] T. Loukopoulos, I. Ahmad, and D. Papadias. An overview of data replication on the Internet. In *Proc. Int. Symp. on Parallel Architectures, Algorithms and Networks ISPAN'02.* IEEE Computer Society Press, 2002.

[92] S. H. Low. A Duality Model of TCP and Queue Management Algorithms. *IEEE/ACM Trans. Networking*, 4(11):525–536, 2003.

[93] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 491–502, 2003.

[94] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop.* IEEE Computer Society Press, 1998.

[95] M. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets.* John Wiley & Sons, 1995.

[96] L. Massoulié and J. Roberts. Bandwidth sharing: Objectives and algorithms. *Transactions on Networking*, 10(3):320–328, June 2002.

[97] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An Architecture for Internet-scale Sensing Services. In *Proceedings of Very Large Databases (VLDB)*, 2003.

[98] B. Olstad and F. Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[99] M. Ouzzani and A. Bouguettaya. Query processing and optimization on the Web. *Distributed and Parallel Databases*, 15(3):187–218, 2004.

[100] V. Pandit and H. Ji. Efficient in-network evaluation of multiple queries. In *Proceedings of HiPC*, pages 205–216, 2006.

[101] P. Pietzuch, J. Leflie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–60, 2006.

[102] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *J. Parallel Distributed Computing*, 64(8):974–996, 2004.

[103] B. Plale and K. Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, 2003.

[104] B. Plateau. *De l'Evaluation du parallélisme et de la synchronisation.* PhD thesis, Université de Paris XII, Orsay (France), 1984.

[105] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the Placement of Web Server Replicas. In *INFOCOM*, pages 1587–1596. IEEE Computer Society Press, 2001.

[106] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer Verlag, 2002.

[107] A. Rosenberg. Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload. *IEEE Trans. Parallel and Distributed Systems*, 13(2):179–191, 2002.

[108] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par'04: Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[109] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

[110] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Science Press, 1995.

[111] O. Sinnen and L. Sousa. Comparison of contention-aware list scheduling heuristics for cluster computing. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 382–387. IEEE Computer Society Press, 2001.

[112] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the grid. In *2002 ACM/IEEE Supercomputing Conference*. ACM Press, 2002.

[113] U. Srivastava, K. Munagala, and J. Widom. Operator Placement for In-Network Stream Query Processing. In *PODS'05: Proceedings of the 24th ACM Symposium on Principles of Database Systems*, pages 250–258. ACM Press, 2005.

[114] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over Web services. In *VLDB '06: Proceedings of the 32nd Int. Conference on Very Large Data Bases*, pages 355–366. VLDB Endowment, 2006.

[115] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 134–143. ACM Press, 1995.

[116] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 62–71. ACM Press, 1996.

[117] X. Tang and J. Xu. QoS-Aware Replica Placement for Content Distribution. *IEEE Trans. Parallel Distributed Systems*, 16(10):921–932, 2005.

[118] K. Taura and A. A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 2000.

[119] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[120] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1999.

[121] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[122] R. van Rennesse, K. Birman, D. Dumitriu, and W. Vogels. Scalable Management and Data Mining Using Astrolabe. In *Proceedings from the First International Workshop on Peer-to-Peer Systems*, pages 280–294, 2002.

[123] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Saddayappan, and J. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Proceedings of Euro-Par'07: Parallel Processing*, LNCS 4641, pages 173–183. Springer Verlag, 2007.

[124] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Saddayappan, and J. Saltz. A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows. In *ICPP'2008, the Int. Conf. on Parallel Processing*, pages 254–261. IEEE Computer Society Press, 2008.

[125] C.-M. Wang, C.-C. Hsu, P. Liu, H.-M. Chen, and J.-J. Wu. Optimizing Server Placement in Hierarchical Grid Environments. In *First International Conference on Grid and Pervasive Computing GPC'07*, LNCS 1900, pages 1–11. Springer, 2007.

[126] H. Wang, P. Liu, , and J.-J. Wu. A QoS-aware Heuristic Algorithm for Replica Placement. In *Proceedings of the 7th International Conference on Grid Computing (GRID2006)*, pages 96–103. IEEE Computer Society Press, 2006.

[127] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, 1991.

[128] R. Wolski, N. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[129] Q. Wu and Y. Gu. Supporting distributed application workflows in heterogeneous computing environments. In *14th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2008.

[130] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, 1994.

[131] W. Yu. *The two-machine flow shop problem with delays and the one-machine total tardiness problem.* PhD thesis, Technishe Universiteit Eidhoven, June 1996.

[132] W. Yu, H. Hoogeveen, and J. K. Lenstra. Minimizing makespan in a two-machine flow shop with delays and unit-time operations is NP-hard. *J. Scheduling*, 7(5):333–348, 2004.

[133] D. Zhu, R. Melhem, and B. Childers. Power-aware scheduling for multiprocessor real-time systems. *IEEE Trans. Parallel Distributed Systems*, 14(7):686–700, 2003.

II Publication list

Note that all my research reports are all available at `graal.ens-lyon.fr/~abenoit/`.

II.1 Journals

[J1] A. Benoit, P. Fernandes, B. Plateau, W.J. Stewart. **On the Benefits of Using Functional Transitions in Kronecker Modelling.** *Performance Evaluation*, 58(4):367-390, Elsevier Science, 2004.

[J2] A. Benoit, L. Brenner, P. Fernandes, B. Plateau. **Aggregation of Stochastic Automata Networks with replicas.** *Linear Algebra and its Applications (LAA)*, 386:111-136, 2004.

[J3] A. Benoit, M. Cole, S. Gilmore, J. Hillston. **Scheduling skeleton-based grid applications using PEPA and NWS**. *The Computer Journal*, special issue on Grid Performability Modelling and Measurement, 48(3):369-378, 2005.

[J4] A. Benoit, M. Cole, S. Gilmore, J. Hillston. **Evaluating the performance of pipeline-structured parallel programs with skeletons and process algebra.** *Scalable Computing: Practice and Experience (SCPE)*, special issue on Practical Aspects of High-level Parallel Programming PAPP 2004, 6(4):1-16, 2005.

[J5] A. Benoit, B. Plateau, W.J. Stewart. **Réseaux d'automates stochastiques à temps discret.** Revue des sciences et technologies de l'information, Technique et science informatiques "Evaluation de Performances", 24(2-3):229-248, 2005.

[J6] A. Benoit, B. Plateau, W.J. Stewart. **Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems.** *Future Generation Computer Systems (FGCS)*, special issue on System Performance Analysis and Evaluation, 22(7):838-847, 2006.

[J7] M. Aldinucci, A. Benoit. **Automatic mapping of ASSIST applications using process algrebra**. *Parallel Processing Letters (PPL)*, 18(1):175-188, 2008.

[J8] A. Benoit, V. Rehn-Sonigo, Y. Robert. **Replica Placement and Access Policies in Tree Networks**. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(12):1614-1627, 2008.

[J9] A. Benoit, Y. Robert. **Mapping pipeline skeletons onto heterogeneous platforms**. *Journal of Parallel and Distributed Computing (JPDC)*, 68(6):790-808, Elsevier, 2008.

[J10] A. Benoit, Y. Robert. **Complexity Results for Throughput and Latency Optimization of Replicated and Data-parallel Workflows**. *Algorithmica*, available online, Springer, October 2008.

[J11] A. Benoit, M. Hakem, Y. Robert. **Contention Awareness and Fault Tolerant Scheduling for Precedence Constrained Tasks in Heterogeneous Systems.** *Parallel Computing (ParCo)*, 35(2):83-108, Elsevier, 2009.

[J12] A. Benoit, H. Kosch, V. Rehn-Sonigo, Y. Robert. **Multi-criteria Scheduling**

of Pipeline Workflows (and Application to the JPEG Encoder). *The International Journal of High Computing Applications (IJHPCA)*, 23:171-187, 2009.

[J13] A. Benoit, L. Marchal, J.F. Pineau, Y. Robert, F. Vivien. **Offline and online scheduling of concurrent bag-of-tasks applications on hetereogeneous platforms.** To appear in *IEEE Transactions on Computers (TC).* Accepted May 2009.

[J14] A. Benoit, Y. Robert, E. Thierry. **On the Complexity of Mapping Linear Chain Applications onto Heterogeneous Platforms.** To appear in *Parallel Processing Letters (PPL).* Accepted March 2009. (Also available as Research report RR-LIP-2008-32, October 2008).

[J15] A. Benoit, M. Hakem, Y. Robert. **Multi-criteria scheduling of precedence task graphs on heterogeneous platforms.** To appear in *The Computer Journal.* Accepted July 2009.

II.2 Conferences

[C1] A. Benoit, B. Plateau, W.J. Stewart. **Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems.** In Proceedings of PMEO-PDS'03, the Int. Workshop on Performance Modelling, Evaluation, and Optimization of Parallel and Distributed Systems (colocated with IPDPS'03). IEEE Computer Society Press, Nice, France, April 2003.

[C2] A. Benoit, L. Brenner, P. Fernandes, B. Plateau. **Aggregation of Stochastic Automata Networks with replicas.** In A.N. Langville, W.J.Stewart (eds), Proceedings of NSMC'03, the 4th Int. Conf. on the Numerical Solution of Markov Chains, pages 145-166, Urbana, Illinois, USA, September 2003.

[C3] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, W.J. Stewart. **The PEPS Software Tool.** In P. Kemper, W.H. Sanders (eds), Proceedings of TOOLS'03, the 13th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, LNCS 2794 pages 98-115, Urbana, Illinois, USA, September 2003.

[C4] A. Benoit, M. Cole, S. Gilmore, J. Hillston. **Evaluating the performance of skeleton-based high level parallel programs.** In M. Bubak, D. van Albada, P. Sloot and J. Dongarra (eds), Proceedings of ICCS'04, the Int. Conf. on Computational Science, Part III, LNCS, pages 299-306. Springer Verlag, 2004.

[C5] A. Benoit, M. Cole, S. Gilmore, J. Hillston. **Analyse quantitative de programmes applicatifs à base de squelettes algorithmiques**. In Proceedings of JFLA'05, Seizièmes Journées Francophones des Langages Applicatifs, Obernai, March 2005.

[C6] A. Benoit, M. Cole, S. Gilmore, J. Hillston. **Enhancing the effective utilisation of Grid clusters by exploiting on-line performability analysis**. In Proceedings of the *First Workshop on Grid Performability*, (colocated with *CCGrid*), Cardiff, UK, May 2005.

[C7] A. Benoit, M. Cole. **Two Fundamental Concepts in Skeletal Parallel Programming**. In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, Proceedings of ICCS'05, the Int. Conf. on Computational Science, Part II, LNCS vol. 3515, pages 764-771. Springer Verlag, 2005.

[C8] M. Aldinucci, A. Benoit. **Automatic mapping of ASSIST applications using process algrebra.** In Proceedings of HLPP'05, Warwick University, Coventry, United Kingdom, July 2005.

[C9] J. Duennweber, A. Benoit, M. Cole, S. Gorlatch. **Native Services for Grid Applications.** In Proceedings of ParCo'05, Malaga, Spain, September 2005.

[C10] A. Benoit, J.P. Chick. **Computer Simulation of the Acoustic Impedance of Modern Orchestra Horns.** In Proceedings of ParCo'05, Malaga, Spain, September 2005.

[C11] A. Benoit, M. Cole, J. Hillston, S. Gilmore. **Using eSkel to implement the multiple baseline stereo application.** In Proceedings of ParCo'05, Malaga, Spain, September 2005.

[C12] A. Benoit, M. Cole, J. Hillston, S. Gilmore. **Flexible Skeletal Programming with eSkel.** In Proceedings of EuroPar'05, the 11th Int. EuroPar Conf. in Lisbon, Portugal. LNCS vol. 3648, Pages 761-770. Springer Verlag. August 2005.

[C13] M. Aldinucci, A. Benoit. **Towards the Automatic Mapping of ASSIST applications for the Grid.** In Proceedings of the CoreGRID Integration Workshop, University of Pisa, Italy, November 2005.

[C14] A. Benoit, V. Rehn, Y. Robert. **Strategies for replica placement in tree networks.** In Proceedings of HCW'07, the 16th Heterogeneity in Computing Workshop, Long Beach, California, USA, March 2007, IEEE Computer Society Press. (Also available as Research report RR-LIP-2006-30, November 2006).

[C15] A. Benoit, Y. Robert. **Mapping pipeline skeletons onto heterogeneous platforms.** In Y. Shi, D. van Albada, J. Dongarra and P. Sloot, editors, Proceedings of ICCS'07, the Int. Conf. on Computational Science, LNCS vol. 4487, pages 591-598. Springer, 2007. (Also avaiblable as Research report RR-LIP-2007-05, January 2007).

[C16] A. Benoit, V. Rehn, Y. Robert. **Impact of QoS on replica placement in tree networks.** In Y. Shi, D. van Albada, J. Dongarra and P. Sloot, editors, Proceedings of ICCS'07, the Int. Conf. on Computational Science, LNCS vol. 4487, pages 366-373. Springer, 2007. (Also avaiblable as Research report RR-LIP-2006-48, December 2006).

[C17] A. Benoit, Y. Robert. **Complexity results for throughput and latency optimization of replicated and data-parallel workflows**. In Proceedings of HeteroPar'07, the 6th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Austin, USA, September 2007. (Also available as Research report RR-LIP-2007-12, March 2007).

[C18] A. Benoit, V. Rehn-Sonigo, Y. Robert. **Multi-criteria scheduling of pipeline workflows**. In Proceedings of HeteroPar'07, the 6th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous

Networks, Austin, USA, September 2007. (Also available as Research report RR-LIP-2007-32, June 2007).

[C19] A. Benoit, V. Rehn-Sonigo, Y. Robert. **Optimizing latency and reliability of pipeline workflow applications**. In Proceedings of HCW'08, the 17th Int. Heterogeneity in Computing Workshop, Miami, USA, April 2008, IEEE Computer Society Press. (Also available as Research report RR-LIP-2008-12, March 2008).

[C20] A. Benoit, L. Marchal, J.F. Pineau, Y. Robert, F. Vivien. **Offline and online scheduling of concurrent bags-of-tasks on heterogeneous platforms**. In Proceedings of APDCM'08, the 10th Workshop on Advances in Parallel and Distributed Computational Models, Miami, USA, April 2008. (Also available as Research report RR-LIP-2007-48, December 2007) .

[C21] A. Benoit, H. Kosch, V. Rehn-Sonigo, Y. Robert. **Bi-criteria pipeline mappings for parallel image processing**. In Proceedings of ICCS'08, the Int. Conf. on Computational Science. LNCS Springer, Krakow, Poland, June 2008. (Also available as Research report RR-LIP-2008-02, January 2008).

[C22] A. Benoit, M. Hakem, Y. Robert. **Fault tolerant scheduling of precedence task graphs on heterogeneous platforms**. In Proceedings of APDCM'08, the 10th Workshop on Advances in Parallel and Distributed Computational Models, Miami, USA, April 2008. (Also available as Research report RR-LIP-2008-03, January 2008).

[C23] A. Benoit, M. Hakem, Y. Robert. **Realistic models and efficient algorithms for fault tolerant scheduling on heterogeneous platforms**. In Proceedings of ICPP'08, the 37th Int. Conf. on Parallel Processing, Portland, USA, September 2008. (Also available as Research report RR-LIP-2008-09, January 2008).

[C24] K. Agrawal, A. Benoit, Y. Robert. **Mapping Linear Workflows with Computation-Communication Overlap**. In Proceedings of ICPADS'08, the 14th IEEE Int. Conf. on Parallel and Distributed Systems, pages 195-202, IEEE Computer Society Press. (Also available as Research report RR-LIP-2008-21, June 2008).

[C25] A. Benoit, Y. Robert, A. Rosenberg, F. Vivien. **Static Strategies for Worksharing with Unrecoverable Interruptions.** In Proceedings of IPDPS'09, the 23rd IEEE Int. Parallel and Distributed Processing Symposium, Rome, Italy, May 2009. (Also available as Research report RR-LIP-2008-29, October 2008).

[C26] A. Benoit, F. Dufossé, Y. Robert. **On the Complexity of Mapping Pipelined Filtering Services on Heterogeneous Platforms.** In Proceedings of IPDPS'09, the 23rd IEEE Int. Parallel and Distributed Processing Symposium, Rome, Italy, May 2009. (Also available as Research report RR-LIP-2008-30, October 2008).

[C27] A. Benoit, L. Marchal, J.F. Pineau, Y. Robert, F. Vivien. **Resource-aware allocation strategies for divisible loads on large-scale systems.** In Proceedings of HCW'09, the 18th Int. Heterogeneity in Computing Workshop,

Rome, Italy, May 2009, IEEE Computer Society Press.

[C28] A. Benoit, H. Casanova, V. Rehn-Sonigo, Y. Robert. **Resource Allocation Strategies for In-Network Stream Processing.** In Proceedings of APDCM'09, the 11th Workshop on Advances in Parallel and Distributed Computational Models, Rome, Italy, May 2009, IEEE Computer Society Press. (Also available as Research report RR-LIP-2008-20, June 2008).

[C29] A. Benoit, F. Dufossé, Y. Robert. **Filter placement on a pipelined architecture.** In Proceedings of APDCM'09, the 11th Workshop on Advances in Parallel and Distributed Computational Models, Rome, Italy, May 2009, IEEE Computer Society Press.

[C30] A. Benoit, A. Dobrila, J.M. Nicod, L. Philippe. **Throughput optimization for micro-factories subject to failures.** In Proceedings of ISPDC'09, the 8th Int. Symp. on Parallel and Distributed Computing, Lisbon, Portugal, July 2009. (Also available as Research report RR-LIP-2009-02, January 2009).

[C31] A. Benoit, Y. Robert. **Multi-criteria Mapping Techniques for Pipeline Workflows on Heterogeneous Platforms.** In *Recent developments in Grid Technology and Applications.* Nova Science Publishers, 2009. To appear.

[C32] K. Agrawal, A. Benoit, F. Dufossé, Y. Robert. **Mapping Filtering Streaming Applications with Communication Costs.** To appear in SPAA'09, Calgary, Canada, August 2009. (Also available as Research report RR-LIP-2009-06, February 2009).

[C33] A. Benoit. **Comparison of Access Policies for Replica Placement in Tree Networks.** To appear in Europar'09, Delft, the Netherlands, August 2009. (Also available as Research report RR-LIP-2009-12, April 2009).

[C34] A. Benoit, M. Gallet, B. Gaujal, Y. Robert. **Computing the throughput of replicated workflows on heterogeneous platforms.** To appear in ICPP'09, Vienna, Austria, September 2009. (Also available as Research report RR-LIP-2009-08, February 2009).

[C35] A. Benoit, M. Hakem, Y. Robert. **Optimizing the Latency of Streaming Applications under Throughput and Reliability Constraints.** To appear in ICPP'09, Vienna, Austria, September 2009. (Also available as Research report RR-LIP-2009-13, April 2009).

[C36] Y. Gu, Q. Wu, A. Benoit, Y. Robert. **Complexity Analysis and Algorithmic Design for Pipeline Configuration in Distributed Networks.** To appear in PODC'09 as a brief announcement, Calgary, Canada, August 2009.

[C37] A. Benoit, H. Casanova, V. Rehn-Sonigo, Y. Robert. **Resource allocation for multiple concurrent in-network stream-processing applications.** To appear in HeteroPar'09, Delft, the Netherlands, August 2009. (Also available as Research report RR-LIP-2009-07, February 2009).

[C38] A. Benoit, Y. Robert, A. Rosenberg, F. Vivien. **Static Worksharing Strategies for Heterogeneous Computers with Unrecoverable Failures.** To appear in HeteroPar'09, Delft, the Netherlands, August 2009.

APPENDIX B

Curriculum vitae

Anne Benoit
Assistant Professor

Laboratoire de l'Informatique et du Parallélisme (LIP)
Ecole Normale Supérieure (ENS) de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France

Phone: +33 (0) 47272 8758, +33 (0) 67621 1008
E-mail: `Anne.Benoit@ens-lyon.fr`
Homepage: `http://graal.ens-lyon.fr/~abenoit`

I Professional experience and education

- **Sept 2005-Present: Assistant Professor**, LIP, Ecole Normale Supérieure de Lyon, France. **Graal** project: *Algorithms and Scheduling for Distributed Heterogeneous Platforms.*

- **Sept 2003-Aug 2005: Research Assistant**, School of Informatics, University of Edinburgh, Scotland. **Enhance** project: *Enhancing the Performance Predictability of Grid Applications with Patterns and Process Algebras*, with Murray Cole, Jane Hillston and Stephen Gilmore.

- **Oct 2000-Sept 2003: PhD in Computer science: Systems and Communications** in the Institut National Polytechnique de Grenoble, defended the 18th June 2003, entitled *Methods and Algorithms for the performance evaluation of systems with a large state space.* Supervisor: Brigitte Plateau (ID-IMAG laboratory, Grenoble, France).

- **1999-2000: "Diplôme d'Etudes approfondies"** (DEA) of Computer science: Systems and Communication (prerequisite for the Ph.D); passed with "Excellent"; University Joseph Fourier, Grenoble, France. **DEA (master) project**, ID-IMAG laboratory, under the supervision of Jacques Chassin de Kergommeaux: *Study of interactions between the Athapascan-0 tracing tool and the traced applications.*

- **1997-2000: École Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble (ENSIMAG)**, INPG, Grenoble, France. Equivalent to a Master's Degree in Computer Science, Emphasis in Systems

and Networks; passed with “Excellent”.
Summer 1999: 8 weeks training period at Caltech (California Institute of Technology), Pasadena, USA.
Summer 1998: 6 weeks training period at IRAM (Institute for Research in Millimeter Astronomy), Grenoble, France.

II Research projects

- **2009-2010:** Responsible of the Lyon partner in the **ALEAE** ARC INRIA project. The goal if this project is to provide models and algorithms in the field of resource management that cope with uncertainties in large-scale distributed systems.

- **2008-2010:** Co-project investigator of the **StochaGrid** ANR project: design of new stochastic models that will allow for an accurate prediction of the performance of workflow applications on grid computing platforms, and new scheduling algorithms.

- **2007-2009:** Project leader of the **SchedLife** CNRS/USA project: “Symbiotic scheduling of biological grid applications”.

- **2006-2009:** Member of the **Alpage** ANR project: “Algorithms for large-scale platforms”.

- **2004-2008:** Member of **CoreGRID**, European Network of Excellence.

- **2003-2005:** Research assistant as part of the **Enhance** project: “Enhancing the performance predictability of grid applications with patterns and process algebras”.

- **2002-2003: DECORE** IMAG-ELESA project: task coordinator “Exploiting the symmetries in communication network models”.

III Teaching

- **Sept 2006-Present:** Responsible of 3rd year computer science students at ENS (Ecole Normale Supérieure de Lyon, France).

- **Sept 2005-Present:** Teaching Algorithms, Parallel Algorithms, and Algorithms for networks and telecommunications at ENS Lyon, and tutoring students.

- **2004-2005:** Tutoring on an Enterprise Computing course at the University of Edinburgh.

- **Oct 2000-Sept 2003:** Gave computer science lectures at ESISAR (INPG, Valence, France) and at ENSIMAG (INPG, Grenoble, France): algorithms, compilation, performance evaluation.

IV Student advising

Post-doctoral students

- **2007-2008:** Co-advising Mourad Hakem with Yves Robert, on the multi-criteria scheduling of precedence task graphs in heterogeneous systems with realistic communication models. See Chapter 5, Section II.
  Joint publications: [J11],[J15],[C22],[C23],[C35].
  Mourad is now assistant professor at IUT Belfort.

PhD students

- **2008-Present:** Co-advising Fanny Dufossé's PhD thesis with Yves Robert, on the mapping of filtering tasks onto large-scale heterogeneous platforms. See Chapter 5, Section I.
  Joint publications: [C26],[C29],[C32].

- **2006-Present:** Co-advising Veronika Rehn-Sonigo's PhD thesis with Yves Robert, on the multi-criteria mapping and scheduling of workflow applications onto heterogeneous platforms. Veronika defended her thesis on July 7, 2009. See Chapter 3, and Chapter 5, Sections III and IV.
  Joint publications: [J8],[J12],[C14],[C16],[C18],[C19],[C21],[C28],[C37].

Master students

- **2009:** Paul Renaud-Goud: mapping of concurrent application workflows onto heterogeneous platforms.

- **2009:** Loïc Magnan: finding efficient operation lists for bi-criteria optimization problems, given a mapping of a workflow application.

- **2008:** Fanny Dufossé: her master's project was the beginning of her PhD thesis, see paragraph above.

- **2003:** Ihab Sbeity: project on translating UML models (system verification) in stochastic automata networks.

V Other professional activities

- **Conference chairing:** Program chair of the HCW 2010 workshop (19th Int. Heterogeneity in Computing Workshop); co-organizing the workshop "Scheduling for large-scale systems" in Knoxville, USA, in May 2009; co-organizing the

PAPP Workshops (Practical Aspects of High-Level Parallel Programming) in 2006, 2007, 2008, 2009, co-located with ICCS (Int. Conf. on Computational Science).

- **Conference program committees:** PC member for the ICCS conference from 2005 to 2009 (Int. Conf. on Computational Science), IPDPS 2008 (Int. Parallel and Distributed Processing Symp.), SBAC-PAD 2008 (Int. Symp. on Computer Architecture and High Performance Computing), HPCC 2009 (Int. Conf. on High Performance Computing and Communications), ISPDC 2009 (Int. Symp. on Parallel and Distributed Computing), ISCIS 2009 (Int. Symp. on Computer and Information Sciences), TCPP PhD Forum 2009 (in conjunction with IPDPS 2009).
- **Reviewing:** Reviewer of several international journals, conferences and workshops, such as IEEE TPDS, IEEE TSE, JPDC, PPL, CCPE, ITOR, IJHPCA, Parco, IPDPS, HCW, ICCS, PAPP, HPCC, ISPDC, EuroPar, ISCIS, ICPADS, SBAC-PAD, PNPM, NSMC, CMPP, CC, Markov Anniversary Proceedings, ...

VI Awards

- **2007-2011: Prime d'encadrement doctoral et de recherche.**
- **Feb 2005: Outstanding thesis award** by the Institut National Polytechnique de Grenoble for my PhD thesis.