N° d'ordre : — *N° attribué par la bibliothèque : ———*

- École normale supérieure de Lyon -
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École normale supérieure de Lyon
Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 20 octobre 2009 par

Matthieu GALLET

Steady-State Scheduling of Task Graphs onto Heterogeneous Platforms

Directeur de thèse :	Frédéric	VIVIEN	
Co-encadrant de thèse :	Yves	ROBERT	
Après avis de :	Arnold	ROSENBERG	Rapporteur
	Olivier	BEAUMONT	Rapporteur
Devant la commission d'examen formée de :			
	Claire	HANEN	Membre
	Arnold	ROSENBERG	Rapporteur
	Olivier	BEAUMONT	Rapporteur
	Jean-François	MÉHAUT	Membre
	Yves	ROBERT	Membre
	Frédéric	VIVIEN	Membre

Contents

Introduction

Since their origins during the Second World War, computers have become faster and faster, more and more powerful. The first machines were slow, like the Zuse's Z3 clocked at 10 Hz, the Colossus clocked at 5.8 MHz or the ENIAC at 100 kHz, but their clock frequency and their overall power quickly increased up to $3-4$ GHz during the first years of the twenty-first century. Since a few years, the clock frequency remains stable but the computing power is still growing as processors can execute more instructions during a single clock cycle.

Despite this constant increase of processing capacity, there were still people asking for more power, and a single processor computer was not powerful enough to fulfill their requirements. The first step to dramatically increase the computing power was made by assembling several processors around the same memory resource shared by all processors. This introduced new algorithmic challenges, mainly due to simultaneous accesses to the same data. This also led to new results, like the famous Amdahl's law giving an upper bound on the theoretical speed-up that can be reached using multiple processors [6].

Since the number of processors in the same computer and using the same memory is limited by hardware constraints, the next step was to gather some computers to form a cluster, each processor having its own private memory. Since the programmer has to take care of communicating data from a processor to another one, clusters bring further algorithmic problems, since communication times and latencies need to be taken into account during the algorithm design phase. Thanks to the deployment of large-scale networks, public ones like the internet or dedicated ones, several clusters can be inter-connected to form a computing grid [44]. At the same time, some specialized supercomputers were replaced by large clusters of simpler but cheaper personal computers, and the multi-core technology was progressively added to processors, even into mainstream processors. Nowadays, the high-end computing clusters are built around processors assisted by graphic cards and heterogeneous multi-core processors, and while dedicated computing structures made of several hundreds or thousands of processors are not uncommon, while public computing grid can reach tens of thousands of processors. In addition of these dedicated structures, more and more users offer idle time of their own desktop computers to form a *desktop grid*, made of numerous but strongly different computers.

From an algorithmic point of view, the models used for the computing platforms became more and more complex, starting from a single processor to many unrelated processors linked by heterogeneous communication links, as summed up by Figure 1. However, this evolution was necessitated by the corresponding hardware evolution. To efficiently handle such large platforms, programs are cut into several tasks, that have to be scheduled. These tasks can be independent, or can have several dependencies, in which case they form a task graph. Thus, the role of scheduling algorithms is to decide where (on which processors) and when to execute these tasks, while taking care of the communications. The goal of such algorithms is to use the hardware as efficiently as possible. However, this notion needs to be carefully explained to define

the objective function of the algorithms. The most classical, and perhaps the most intuitive, objective function is to minimize the *makespan*, or, in other words, the overall time necessary for the platform to achieve all computations.

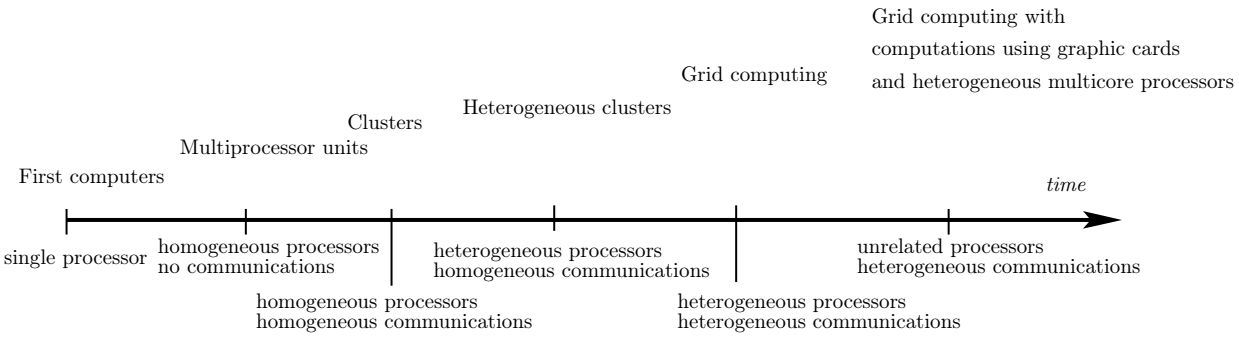


Figure 1: Short summary of the evolution of platform models.

However, when minimizing the makespan, simple scheduling problems are NP-hard, even in the context of homogeneous resources [30]: if we have only two identical processors and no communications, minimizing the completion time of several jobs is already NP-hard [65]. Furthermore, this objective function is not adapted to all scheduling problems, especially when the user wants to schedule a continuous flow of tasks (the makespan is undefined in this case). If this flow is made of a sequence of identical computations applied to hundreds or thousands of data sets, then one could prefer to maximize the *throughput* of the platform, i.e., the average number of data sets which are executed by the platform every time unit. Considering the average throughput of the platform instead of the makespan leads to focus on the heart of the schedule, leaving apart the matter of optimizing both the initialization and the termination of the execution. If the number of tasks is large enough, we hope that optimizing the *steady-state* is enough to ensure an efficient use of the platform.

This thesis is focused on scheduling large numbers of independent jobs, these jobs being atomic or, on the contrary, being split into several tasks with inter-task dependencies. Minimizing the total completion time is a hard problem, and thus we have to relax some constraints to keep the problem tractable. We concentrate on two different relaxations: the Divisible Load Theory and the Steady-State scheduling.

Divisible Load Theory

This first part is devoted to the Divisible Load Theory, a classical relaxation of makespan minimization problems. In Chapter 1, we present this relaxation through the schedule of an example of application on a star-shaped platform, using several familiar results. The essence of the Divisible Load Theory is to consider a single load, which can be arbitrarily divided into several chunks. We also discuss the use of several installments to distribute the entire load to workers and the introduction of latencies in the communication model, instead of a simple linear cost model.

In Chapter 2, we consider a linear network of heterogeneous processors and several independent divisible loads, each load having its own characteristics in terms of communication data and amount of computation. All loads are initially stored on the first processor of the chain, which keeps a fraction of each load and redistributes the remaining to the second processor. As

soon as this second processor has received its fraction of a load, it keeps a fraction of it and forwards the remaining to the third processor, and so on along the processor chain. The whole framework is fully described in Section 2.2. This problem was first studied by Min, Veeravalli, and Barlas in [84, 85]. However, their solution is not optimal, or even not valid. Thus, we expose a valid, and optimal, solution to this problem in Section 2.4, before comparing all strategies in Section 2.6. This work has been published in [C3], in [A1], and in [A2].

Steady-State scheduling

In this second part, we turn our attention to the schedule of many instances of the same task or the same task graph on a heterogeneous platform. Instead of minimizing the makespan, which is hard and not really useful for this problem, we rather study the maximization of the platform throughput, without considering the start nor the end of generated schedules.

After a general presentation of the Steady-State techniques in Chapter 3, we study the scheduling of complex but static applications, made of acyclic task graphs, on a heterogeneous platform. However, schedules remain quite simple, made of a single allocation. In a second step (in Chapter 5), we focus on a collection of simpler but dynamic applications: the characteristics of their instances are varying. Designing static schedules taking care of this dynamicity is difficult, even in case of simple bag-of-tasks applications. Chapter 6 deals with pipeline applications, with more complex schedules: several tasks are replicated on several processors to increase the global throughput. In this case, even if instances are distributed in a simple Round-Robin fashion and if the mapping is completely specified, computing the throughput of the platform is difficult. Finally, Steady-State techniques are actually used in Chapter 7, being adapted to a single heterogeneous multi-core processor.

Chapter 3 introduces some notions, which are common to the different chapters constituting this part. We give a general formulation of the problem, explaining the model used for platforms and applications in Section 3.2. In Section 3.3, we give the formal definition of a periodic schedule. In a nutshell, a periodic schedule of period $\mathcal{T}$ is made of a sequence of allocations of tasks to processors which is repeated every $\mathcal{T}$ time units. In Section 3.4, we briefly compare the respective advantages and drawbacks of periodic schedules and of dynamic strategies, which are commonly used on computing grids.

Chapter 4 is centered on providing simple but efficient periodic schedules of complete task graphs on a heterogeneous platform. These schedules are made of a single allocation, i.e., all instances of a given task are mapped on the same processor, in order to decrease the required flow control. Section 4.3 gives an optimal solution to this problem. However, the associated decision problem is proven to be NP-hard, and the complexity of our optimal solution is exponential and cannot be computed for large instances. Thus, we also propose several heuristics in Section 4.4. All these mono-allocation heuristics are compared to two dynamic scheduling strategies in Section 4.5, proving the strong efficiency of our approach. This work has been published in [B3] and in [B1].

Instead of full task graphs, Chapter 5 deals with different bag-of-tasks applications, i.e., applications which are made of several independent tasks. The target computing structure is a star-shaped platform, made of several workers organized around a master processor, which initially owns all data. Applications such that all instances have identical sizes are the most commonly studied. However, there are many applications, such that instances have similar but different characteristics. In Section 5.3, we first recall the solution when all instances of each application share the same characteristics, before proposing an ε-approximation to the *off-line*

problem (all instances are known before the execution of the algorithm) and a heuristic to the *online* problem (instances are progressively submitted to the scheduler). In Section 5.4, some experiments show that our approach offers good results in comparison with dynamic demand-driven policies.

In Chapter 6, we discuss schedules of linear task graphs, or workflows, mapped on a fully connected platform. Contrarily to the previous situations, a given task can be mapped on several processors in a Round-Robin fashion: if a task is mapped on two processors, then the first one processes each even instance of the task, while the second processor process odd ones, independently of their respective speeds. It appears that even if the whole mapping of tasks on processors is given, then determining the period is a complex problem. To solve it, we present in Section 6.3 models based on timed Petri nets. This technique allows us to compute the throughput of the platform in Section 6.4, once the mapping is given. This work has been published in [B2].

Contrarily to the other chapters, which are mostly devoted to large structures like clusters or grids, Chapter 7 is dedicated to a single processor. This processor, the Cell made by IBM, is very different from common multi-core processors because of its two types of cores. Around a classic PowerPC core, we find eight smaller cores called SPEs. If these cores perform very well in vectorial computing, there are less efficient in some other tasks like double precision floating-point operations. In Section 7.2, we describe all the hardware components of the Cell from a theoretical point of view. This heterogeneity is one of the reasons of the complexity of programming for the Cell, especially in the case of multiple instances of the same task graph. Thus, in Chapter 7.3, we apply steady-State techniques using a single allocation to obtain efficient schedules: a complete framework for using such schedules has been implemented, and some experimental comparisons to basic schedules are given in Section 7.4.

Part I

Divisible load theory

Chapter 1

Presentation of the Divisible Load Theory

1.1 Introduction

This first part targets the problem of scheduling a large and compute-intensive application on parallel resources, typically organized as a master-worker platform. We assume that we can arbitrarily split the total work, or load, into chunks of arbitrary sizes, and that we can distribute these chunks to an arbitrary number of workers. The job has to be perfectly parallel, without any dependence between sub-tasks. In practice, this model is a reasonable relaxation of an application made up of a large number of identical, fine-grain parallel computations. Such applications are found in many scientific areas, like linear algebra [32], satellite pictures processing [62], multimedia contents [4, 5] broadcasting, image processing [62, 66], large databases searching [41, 29], biological pattern-matching [64]. This model is known as the *Divisible Load* model and has been widespread by Bharadwaj, Ghose, Mani, and Robertazzi in [28]. Steady-state scheduling, detailed in Part II, is another relaxation, more sophisticated but well-suited to complex applications. In [68], Robertazzi shows that the Divisible Load model is a tractable approach, which applies to a great number of scheduling problems and to a large variety of platforms, such as bus-shaped, star-shaped, and even tree-shaped platforms.

Divisible load theory provides a practical framework for the mapping of independent tasks onto heterogeneous platforms. From a theoretical standpoint, the success of the divisible load model is mostly due to its analytical tractability. Optimal algorithms and closed-form formulas exist for the simplest instances of divisible load problems. We are aware of only one NP-completeness result in the DLT [88]. This is in sharp contrast with the theory of task graph scheduling, which abounds in NP-completeness theorems and in inapproximability results.

In this chapter, we motivate the Divisible Load model using the example of a seismic tomography application processed by a star-shaped platform. We recall to solve this example first with the classical approach in Section 1.1.2, and then using the Divisible Load model in Section 1.2. After the presentation of the complete resolution, we use weaker assumptions to expose more general but harder problems in Section 1.3. Finally, we conclude this chapter in Section 1.4. Chapter 2 applies these DLT techniques to the problem of scheduling several applications on a different type of platforms, based on linear chains of processors. In this second chapter, we present an optimal solution using a given number of installments. We also show that any optimal solution requires an infinite number of rounds.

1.1.1 Motivating Example

We use a specific example of application and platform combination as a guideline, namely an Earth seismic tomography application deployed on a star-shaped platform of processors and presented by Genaud, Giersch and Vivien in [47]. The application is used to validate a model for the internal structure of the Earth, by comparing for every seismic event the propagation time of seismic waves as computed by the model with the time measured by physical devices. Each event is independent from the others and there is a large number of such events: 817,101 events were recorded for the sole year of 1999. The master processor owns all items, reads them, and scatters them among n active workers. Then each worker can process the data received from the master independently. The objective is to minimize the total completion time, also known as the makespan. The simplified code of the program can be written as:

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

This application fulfills all assumptions of the Divisible Load model, since it is made of a very large number of fine grain computations, and these computations are independent. Indeed, we do not have any dependence, synchronization, nor communication between tasks.

Throughout this chapter we will consider two different types of platforms. The first one is a bus-shaped master-worker platform, where all workers are connected to the master through identical links, and the second one is a star-shaped master-worker platform, where workers are connected to the master through links of different characteristics.

1.1.2 Classical Approach

In this section, we aim at solving the problem in a classical way. The target platform is a bus-shaped network, as shown in Figure 1.1. Workers are *a priori* heterogeneous, hence they have different computation speeds. We enforce the full one-port communication model: the master can communicate to at most one worker at a time. This model corresponds to the behavior of two widespread MPI libraries, IBM MPI and MPICH, which serialize asynchronous communications as soon as message sizes exceed a few tens of kilobytes [70]. Finally, each worker receives its whole share of data in a single message.

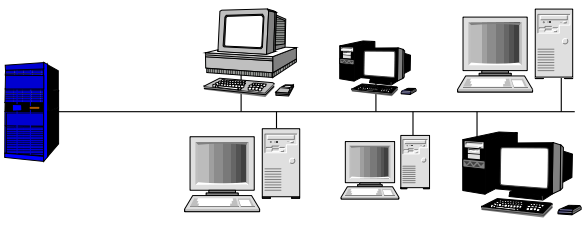


Figure 1.1: Example of bus-shaped network.

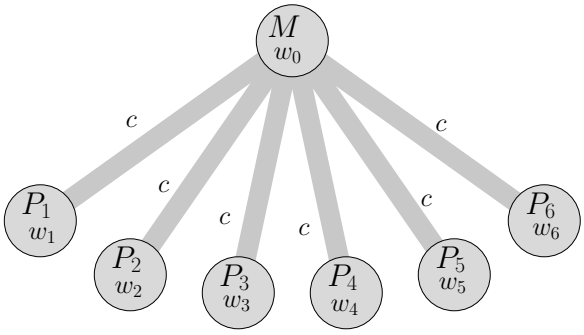


Figure 1.2: Theoretical model of a bus-shaped network.

Figure 1.3: M computes and sends data to P_1.

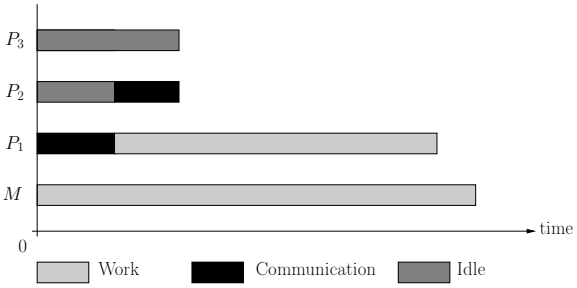


Figure 1.4: M and P_1 compute, M sends data to P_2.

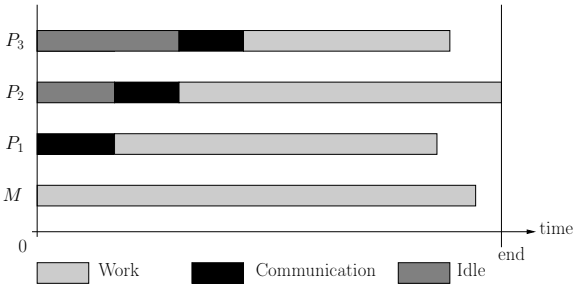


Figure 1.5: Complete schedule.

We will use the following notations, as illustrated in Figure 1.2:

- M is the master processor, which initially holds all the data to process.
- There are n workers, denoted as $P_1, \ldots, P_n$. In order to simplify some equations, P_0 denotes the master processor M.
- Worker P_i takes a time w_i to execute a task. $M = P_0$ requires a time w_0 to process such a load.
- Any worker P_i needs c time units to receive a unit-size load from the master. Recall that all workers communicate at the same speed with the master.
- M initially holds a total load of size W_{total}, where W_{total} is a very large integer.
- M will allocate n_i tasks to worker P_i. n_i is an integer, and since all tasks have to be processed, we have $\sum_{i=0}^{p} n_i = W_{total}$ (we consider that the master can also process some tasks).
- The completion time T_i corresponds to the end of the computations of P_i.

We allow the overlap of communications by computations on the master, i.e., the master can send data to workers while computing its own data. A worker cannot begin its computations before having received all its data. The objective is to minimize the total completion time T needed to compute the load W_{total}. We have to determine values for the n_i's that minimize T. Let us compute the completion time T_i of processor P_i : If we assume that processors are served in the order $P_1, \ldots, P_n$, equations are simple:

- $P_0 : T_0 = n_0 \cdot w_0$,
- $P_1 : T_1 = n_1 \cdot c + n_1 \cdot w_1$,
- $P_2 : T_2 = n_1 \cdot c + n_2 \cdot c + n_2 \cdot w_2$,
- $P_i : T_i = \sum_{j=1}^{i} n_j \cdot c + n_i \cdot w_i$ for $i \geq 1$.

These equations are illustrated by Figures 1.3, 1.4, and 1.5. If we let $c_0 = 0$ and $c_i = c$ for $i \geq 1$, we render the last equation more homogeneous: $T_i = \sum_{j=0}^{i} n_j \cdot c_j + n_i \cdot w_i$ for $i \geq 0$.

By definition, the total completion time T is equal to:

$$T = \max_{0 \leq i \leq n} \left(\sum_{j=0}^{i} n_j \cdot c_j + n_i \cdot w_i \right). \tag{1.1}$$

If we rewrite Equation (1.1) as

$$T = n_0 \cdot c_0 + \max \left(n_0 \cdot w_0, \max_{1 \leq i \leq n} \left(\sum_{j=1}^{i} n_j \cdot c_j + n_i \cdot w_i \right) \right),$$

we recognize an optimal sub-structure for the distribution of $W_{total} - n_0$ tasks among processors P_1 to P_n. This remark allows to easily find a solution for $n_0, \ldots, n_p$ using dynamic programming. Such a solution is given by Algorithm 1 [1], presented in [47].

Algorithm 1: Solution for the classical approach, using dynamic programming

```
solution[0, n] ← cons(0, NIL)
cost[0, n] ← 0
for d ← 1 to W_total do
    solution[d, n] ← cons(d, NIL)
    cost[d, n] ← d · c_n + d · w_n
for i ← n − 1 downto 0 do
    solution[0, i] ← cons(0, solution[0, i + 1])
    cost[0, i] ← 0  for d ← 1 to W_total do
        (sol, min) ← (0, cost[d, i + 1])
        for e ← 1 to d do
            m ← e · c_i + max(e · w_i, cost[d − e, i + 1])
            if m < min then
                (sol, min) ← (e, m)
        solution[d, i] ← cons(sol, solution[d − sol, i + 1])
        cost[d, i] ← min
return (solution[W_total, 0], cost[W_total, 0])
```

Nevertheless, this solution is not really satisfying and suffers from several drawbacks. First, we do not have a closed form solution (neither for the n_i's nor for T). Moreover, the order of the processors during the distribution is arbitrarily fixed (the master communicates with P_1, then with P_2, P_3, and so on). Since processor speeds are *a priori* different, this order could be sub-optimal, and this solution does not help us to find the right order among the $n!$ possible orders. There are by far too many possible orders to try an exhaustive search. Furthermore, the time complexity of this solution is $W_{total}^2 \times n$, so the time to decide for the right values of the n_i's can be greater than the time of the actual computation! Finally, if W_{total} is slightly changed, we cannot reuse the previous solution to obtain a new distribution of the n_i's and we have to redo the entire computation.

Even if we have an algorithm, which gives us a (partial) solution in pseudo-polynomial time, we can still look for a better way to solve the problem. Let us consider the Divisible Load model.

[1] Algorithm 1 builds the solution as a list. Hence the use of the constructor *cons* that adds an element at the head of a list.

If we have around 800,000 tasks for 10 processors, there are on average 80,000 tasks on each processor. So, even if we can have a solution in rational numbers (i.e., a non-integer number of tasks on each processor), we easily afford the extra-cost of rounding this solution to obtain integer numbers of tasks and then a valid solution in practice. The new solution will overcome all previous limitations.

1.2 Divisible Load Approach

The main principle of the Divisible Load model is to relax the integer constraint on the number of tasks on each worker. This simple idea can lead to high-quality results, even if we loose a little precision in the solution. Now, let us instantiate the problem using this relaxation: instead of an integer number n_i of tasks, processor P_i (with $0 \leq i \leq n$) will compute a fraction α_i of the total load W_{total}, where $\alpha_i \in \mathbb{Q}$. The number of tasks allocated to P_i is then equal to $n_i = \alpha_i W_{total}$ and we add the constraint $\sum_{i=0}^{n} \alpha_i = 1$, in order to ensure that the whole workload will be computed.

1.2.1 Bus-Shaped Network

In this paragraph, we keep exactly the same platform model as before: i.e., a bus-shaped network with heterogeneous workers, and data is distributed to workers in a single message, following a linear cost model. Equation (1.1) can easily be translated into Equation (1.2).

$$T = \max_{0 \leq i \leq n} \left(\sum_{j=0}^{i} \alpha_j \cdot c_j + \alpha_i \cdot w_i \right) W_{total}. \tag{1.2}$$

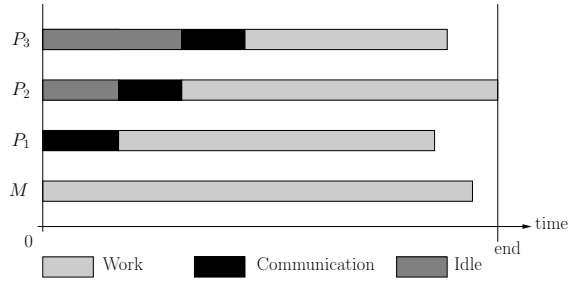
Using this equation, one can prove several important properties of optimal solutions, which were first given in [28]:

- all processors participate ($\forall i, \alpha_i > 0$) and end their work at the same time (Lemma 1.1),
- the master processor should be the fastest computing one but the order of other processors is not important (Lemma 1.2).

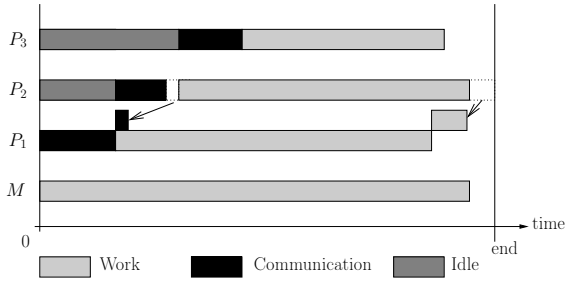
Lemma 1.1. *In an optimal solution, all processors participate and end their processing at the same time.*

Proof. Consider any solution, such that at least two workers do not finish their work at the same time. Without any loss of generality, we can assume that these two processors are P_i and P_{i+1} (with $i \geq 0$) and such that either P_i or P_{i+1} finishes its work at time T (the total completion time). In the original schedule, P_i finishes at time T_i, P_{i+1} finishes at time T_{i+1} and $\max(T_i, T_{i+1}) = T$. We transfer a fraction δ of work from P_{i+1} to P_i (Figure 1.6), assuming that transferring a negative amount of work corresponds to a transfer from P_i to P_{i+1}. In this new schedule, P_i finishes at time T'_i and P_{i+1} finishes at time T'_{i+1}. We want P_i and P_{i+1} to finish at the same time, thus we have the following equations:

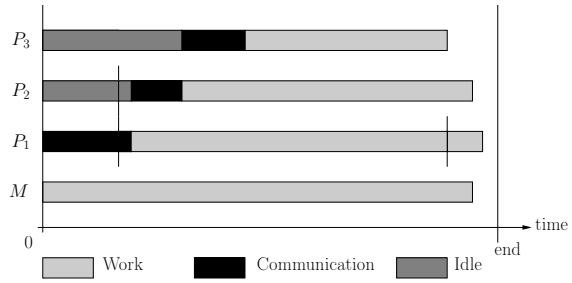
$$\begin{aligned}
& & T'_i &= T'_{i+1} \\
\Leftrightarrow & \left(\sum_{j=0}^{i} \alpha_j c_j\right) + \delta c_i + \alpha_i w_i + \delta w_i &= & \left(\sum_{j=0}^{i} \alpha_j c_j\right) \\
& & & +\delta c_i + (\alpha_{i+1} - \delta)(c_{i+1} + w_{i+1}) \\
\Leftrightarrow & \delta c_i + \alpha_i w_i + \delta w_i &= & \delta c_i + \alpha_{i+1}(c_{i+1} + w_{i+1}) \\
& & & -\delta(c_{i+1} + w_{i+1}) \\
\Leftrightarrow & \delta\left(w_i + c_{i+1} + w_{i+1}\right) &= & \alpha_{i+1}\left(c_{i+1} + w_{i+1}\right) - \alpha_i w_i
\end{aligned}$$



(a) P_1 finishes earlier than P_2.



(b) A fraction of the load allocated to P_2 is given to P_1.



(c) This new schedule is strictly better.

Figure 1.6: Discharging some work from P_2 to P_1 to obtain a better schedule.

Therefore,

$$T'_i = T'_{i+1} \quad \Leftrightarrow \quad \delta = \frac{\alpha_{i+1}\left(c_{i+1} + w_{i+1}\right) - \alpha_i w_i}{w_i + c_{i+1} + w_{i+1}}.$$

We have:

$$\begin{aligned} T'_{i+1} - T_{i+1} &= \delta\left(c_i - c_{i+1} - w_{i+1}\right), \\ T'_i - T_i &= \delta\left(c_i + w_i\right). \end{aligned}$$

We know that $c_{i+1} \geq c_i \geq 0$ (because $c_i = c$ if $i \geq 1$ and $c_0 = 0$). Thus, if δ is positive, then we have $T = T_{i+1} > T'_{i+1} = T'_i > T_i$, and $T = T_i > T'_i = T'_{i+1} > T_{i+1}$ if δ is negative. In both cases, processors P_i and P_{i+1} finish strictly earlier than T. Finally, δ cannot be zero, since it would mean that $T_i = T'_i = T'_{i+1} = T_{i+1}$, contradicting our initial hypothesis.

Moreover, there are at most $n-1$ processors finishing at time T in the original schedule. By applying this transformation to each of them, we exhibit a new schedule, strictly better than the original one. In conclusion, any solution such that all processors do not finish at the same time can be strictly improved, hence the result.

■

Lemma 1.2. *If we can choose the master processor, it should be the fastest processor, but the order of the other processors has no importance.*

Proof. Let us consider any optimal solution $(\alpha_0, \alpha_1, \ldots, \alpha_n)$. By using Lemma 1.1, we know that $T = T_0 = T_1 = \ldots = T_n$. Hence the following equations:

- $T = \alpha_0 \cdot w_0 \cdot W_{total}$,
- $T = \alpha_1 \cdot (c + w_1) \cdot W_{total}$ and then $\alpha_1 = \frac{w_0}{c+w_1}\alpha_0$,
- $T = (\alpha_1 \cdot c + \alpha_2 \cdot (c + w_2))\, W_{total}$ and then $\alpha_2 = \frac{w_1}{c+w_2}\alpha_1$,
- for all $i \geq 1$, we derive $\alpha_i = \frac{w_{i-1}}{c+w_i}\alpha_{i-1}$.

Thus, we have $\alpha_i = \prod_{j=1}^{i} \frac{w_{j-1}}{c+w_j}\alpha_0$ for all $i \geq 0$. Since $\sum_{i=0}^{n} \alpha_i = 1$, we can determine the value of α_0, and thus we have closed formulas for all the α_i's:

$$\alpha_i = \frac{\prod_{j=1}^{i} \frac{w_{j-1}}{c_j+w_j}}{\sum_{k=0}^{n} \left(\prod_{j=1}^{k} \frac{w_{j-1}}{c_j+w_j}\right)}.$$

Using these formulas, we are able to prove the lemma. Let us compute the work done in time T by processors P_i and P_{i+1}, for any i, $0 \leq i \leq n-1$. As in Section 1.1.2, we let $c_0 = 0$ and $c_i = c$ for $1 \leq i \leq n$, in order to have homogeneous equations. Then, the two following equations hold true:

$$T = T_i = \left(\left(\sum_{j=0}^{i-1} \alpha_j \cdot c_j\right) + \alpha_i \cdot w_i + \alpha_i \cdot c_i\right) \cdot W_{total}, \tag{1.3}$$

$$T = T_{i+1} = \left(\left(\sum_{j=0}^{i-1} \alpha_j \cdot c_j\right) + \alpha_i \cdot c_i + \alpha_{i+1} \cdot w_{i+1} + \alpha_{i+1} \cdot c_{i+1}\right) \cdot W_{total}. \tag{1.4}$$

With $K = \frac{T - W_{total} \cdot \left(\sum_{j=0}^{i-1} \alpha_j \cdot c_j\right)}{W_{total}}$, we have

$$\alpha_i = \frac{K}{w_i + c_i} \qquad \text{and} \qquad \alpha_{i+1} = \frac{K - \alpha_i \cdot c_i}{w_{i+1} + c_{i+1}}.$$

The total fraction of work processed by P_i and P_{i+1} is equal to

$$\alpha_i + \alpha_{i+1} = \frac{K}{w_i + c_i} + \frac{K}{w_{i+1} + c_{i+1}} - \frac{c_i \cdot K}{(w_i + c_i)(w_{i+1} + c_{i+1})}.$$

Therefore, if $i \geq 1$, $c_i = c_{i+1} = c$, and this expression is symmetric in w_i and w_{i+1}. We can then conclude that the communication order to the workers has no importance on the quality of a solution, even if it is contrary to the intuition. However, when $i = 0$ the amount of work done becomes:

$$\alpha_0 + \alpha_1 = \frac{K}{w_0} + \frac{K}{w_1 + c}.$$

Since $c > 0$, this value is maximized when w_0 is smaller than w_1. Hence, the root processor should be the fastest computing ones, if possible. ■

The previous analysis can be summarized as:

Theorem 1.1. *For Divisible Load applications on bus-shaped networks, in any optimal solution, the fastest computing processor is the master processor, the order of the communications to the workers has no impact on the quality of a solution, and all processors participate in the work and finish simultaneously. A closed-form formula gives the fraction α_i of the load allocated to each processor:*

$$\forall i \in \{0, \ldots, p\}, \alpha_i = \frac{\prod_{j=1}^{i} \frac{w_{j-1}}{c_j+w_j}}{\sum_{k=0}^{n} \left(\prod_{j=1}^{k} \frac{w_{j-1}}{c_j+w_j}\right)}.$$

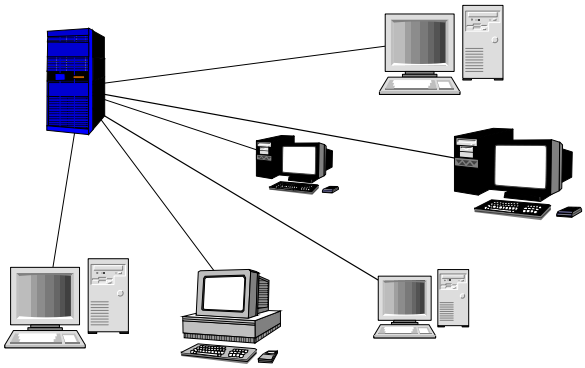


Figure 1.7: Example of star-shaped network.

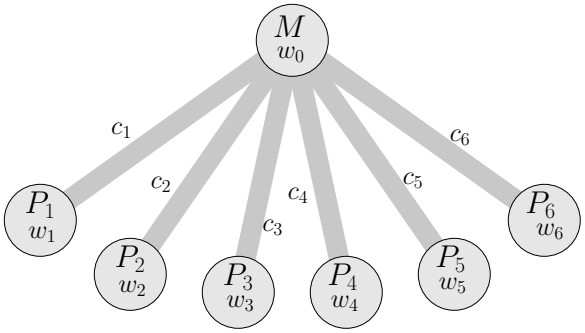


Figure 1.8: Theoretical model of a star-shaped network.

1.2.2 Star-Shaped Network

The bus-shaped network model can be seen as a particular case, with homogeneous communications, of the more general star-shaped network. Now, we focus our attention on such star-shaped networks: every worker P_i is linked to M through a communication link of different capacity as shown in Figure 1.7, and processors have different speeds.

Notations are similar to the previous ones and are illustrated by Figure 1.8: a master processor M, and n workers $P_1, \ldots, P_n$. The master sends a unit-size message to P_i (with $1 \leq i \leq n$) in time c_i, and P_i processes it in time w_i. The total workload is W_{total}, and P_i receives a fraction α_i of this load (with $\alpha_i \in \mathbb{Q}$ and $\sum_{i=1}^{n} \alpha_i = 1$).

We assume that the master does not participate in the computation, because we can always add a virtual processor P_0, with the same speed w_0 as the master and with instantaneous communications: $c_0 = 0$. As before, M sends a single message to each worker, and it can communicate to at most one worker at a time, following the one-port model.

This new model seems to be a simple extension of the previous one, and we will check whether our previous lemmas are still valid. We show that the following lemmas (first shown by Beaumont, Casanova, Legrand, Robert and Yang in [14]) are true for any optimal solution:

- all processors participate in the work (Lemma 1.3),
- all workers end their work at the same time (Lemma 1.4),
- all active workers must be served in the non-decreasing order of the c_i's (Lemma 1.5),
- an optimal solution can be computed in polynomial time using a closed-form expression (Lemma 1.6).

Lemma 1.3. *In any optimal solution, all processors participate in the work, i.e.,* $\forall i, \alpha_i > 0$.

Proof. Consider an optimal solution $(\alpha_1, \ldots, \alpha_p)$ and assume that at least one processor P_i remains idle during the whole computation ($\alpha_i = 0$). Without any loss of generality, we can also assume that communications are served in the order $(P_1, \ldots, P_n)$. We denote by k the greatest index such that $\alpha_k = 0$ (i.e., P_k is the last processor which is kept idle during the computation). We have two cases to consider:

- $k < n$

  By definition, P_n is not kept idle and thus $\alpha_n \neq 0$. We know that P_n is the last processor to communicate with the master, and then there is no communication during the last $\alpha_n \cdot w_n \cdot W_{total}$ time units. Therefore, once P_n as received its share of work, we can send

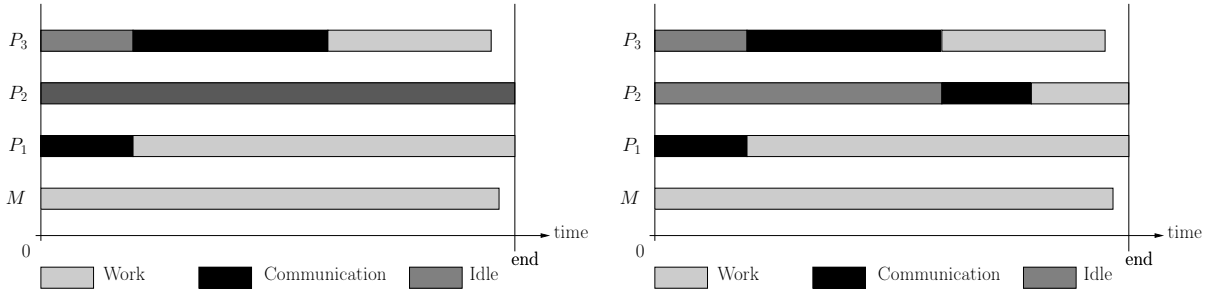


Figure 1.9: Some work is added to P_k.

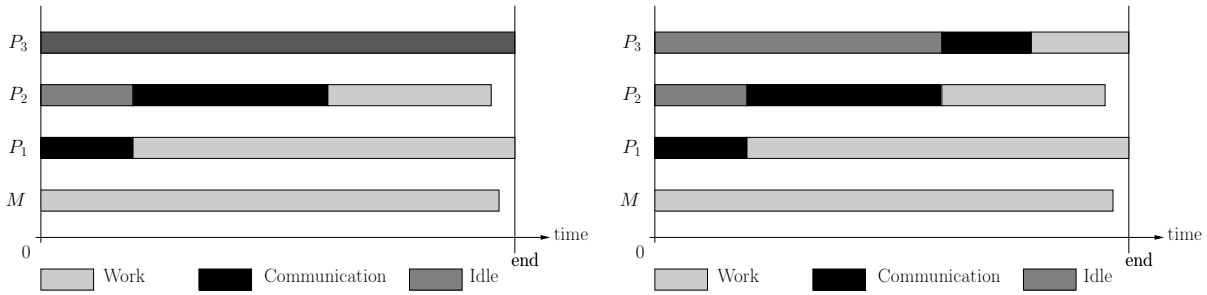


Figure 1.10: Some work is added to P_m.

$\frac{\alpha_n \cdot w_n \cdot W_{total}}{c_k + w_k} > 0$ load units to the processor P_k, as illustrated in Figure 1.9, and it would finish its computation at the same time as P_n.

- $k = n$
  We modify the initial solution to give some work to the last processor, without increasing the total completion time. Let k' be the greatest index such that $\alpha_{k'} \neq 0$. By definition, since $P_{k'}$ is the last served processor, there is no communication with the master during at least $a_{k'}.w_{k'} \cdot W_{total}$ time units. As previously, we can give $\frac{\alpha_{k'} \cdot w_{k'} \cdot W_{total}}{c_n + w_n} > 0$ load units to P_n and then exhibits a strictly better solution than the previous optimal one, as represented by Figure 1.10.

Then, in all cases, if at least one processor remains idle, we can build a solution processing strictly more work within the same time. Then, scaling everything, this leads to a solution processing the same amount of work in strictly less time. This proves that in any optimal solution, all processors participate in the work. ■

Lemma 1.4. *In any optimal solution, all workers end their work at the same time.*

Proof. Consider any optimal allocation. Without loss of generality, we suppose the processors to be numbered according to the order in which they receive their shares of the work. We denote our optimal solution $(\alpha_1, \ldots, \alpha_n)$. Thanks to Lemma 1.3, we know that all workers participate in the computation and then all α_i's are strictly positive. Consider the following linear program (1.5):

$$\left\{\begin{array}{ll} \text{MAXIMIZE } \sum_{i=1}^{n}\beta_i \text{ UNDER THE CONSTRAINTS} & \\ (1.5a) \quad \forall i, \beta_i & \geq 0, \\ (1.5b) \quad \forall i, \sum_{k=1}^{i}\beta_k c_k + \beta_i w_i & \leq T \end{array}\right. \tag{1.5}$$

The α_i's obviously satisfy the set of constraints above, and from any set of β_i's satisfying the set of inequalities, we can build a valid schedule that processes exactly $\sum_{i=1}^{n}\beta_i$ units of load. Let $(\beta_1, \ldots, \beta_n)$ be any optimal solution to this linear program. By definition, we have $\sum_{i=1}^{n}\alpha_i = \sum_{i=1}^{n}\beta_i$.

We know that one of the extremal solutions $\mathcal{S}_1$ of the linear program is one of the vertices of the convex polyhedron $\mathcal{P}$ induced by the inequalities [72]: this means that in the solution $\mathcal{S}_1$, there are at least n equalities among the $2 \cdot n$ inequalities, n being the number of variables. If we use Lemma 1.3, we know that all the β_i's are positive. Then this vertex is the solution of the following (full rank) linear system

$$\forall i \in (1, \ldots, n), \sum_{k=1}^{i}\beta_k c_k + \beta_i w_i = T.$$

Thus, we derive that there exists at least one optimal solution, such that all workers finish simultaneously.

Now, consider any other optimal solution $\mathcal{S}_2 = (\delta_1, \ldots, \delta_n)$, different from $\mathcal{S}_1$. Similarly to $\mathcal{S}_1$, $\mathcal{S}_2$ belongs to the polyhedron $\mathcal{P}$. Now, consider the following function f:

$$f : \left\{\begin{array}{l} \mathbb{R} \rightarrow \mathbb{R}^n \\ x \mapsto \mathcal{S}_1 + x(\mathcal{S}_2 - \mathcal{S}_1) \end{array}\right. .$$

By construction, we know that $\sum_{i=1}^{n}\beta_i = \sum_{i=1}^{n}\delta_i$. Thus, with the notation $f(x) = (\gamma_1(x), \ldots, \gamma_n(x))$:

$$\forall i \in (1, \ldots, n), \gamma_i(x) = \beta_i + x(\delta_i - \beta_i),$$

and

$$\forall x, \sum_{i=1}^{n}\gamma_i(x) = \sum_{i=1}^{n}\beta_i = \sum_{i=1}^{n}\delta_i.$$

Therefore, all the points $f(x)$ that belong to $\mathcal{P}$ are extremal solutions of the linear program.

Since $\mathcal{P}$ is a convex polyhedron and both $\mathcal{S}_1$ and $\mathcal{S}_2$ belong to $\mathcal{P}$, then $\forall x, 0 \leq x \leq 1, f(x) \in \mathcal{P}$. Let us denote by x_0 the largest value of $x \geq 1$ such that $f(x)$ still belongs to $\mathcal{P}$. x_0 is finite, otherwise at least one of the upper bounds or one of the lower bounds would be violated. At least one constraint of the linear program is an equality in $f(x_0)$, and this constraint is not satisfied for $x > x_0$. We know that this constraint cannot be one of the upper bounds: otherwise, this constraint would be an equality along the whole line $(\mathcal{S}_1, f(x_0))$, and would remain an equality for $x > x_0$. Hence, the constraint of interest is one of the lower bounds. In other terms, there exists an index i, such that $\gamma_i(x_0) = 0$. This is in contradiction with Lemma 1.3 and with the fact that the γ_i's correspond to an optimal solution of the problem.

We can conclude that $\mathcal{S}_1 = \mathcal{S}_2$, and thus that for a given order of the processors, there exists a unique optimal solution and that in this solution all workers finish simultaneously their work. ■

Lemma 1.5. *In any optimal solution all active workers must be served in the non-decreasing order of the c_i's.*

Proof. We use the same method as for proving Lemma 1.2: we consider two workers P_i and P_{i+1} (with $1 \leq i \leq n-1$) and we check whether the total work which can be done by them in a given time T is dependent of their order.

For these two processors, we use Lemma 1.4 to obtain the following Equations (1.6) and (1.7), similar to Equations (1.3) and (1.4) derived in the case of a bus-shaped network:

$$T = T_i = \left(\left(\sum_{j=1}^{i-1} \alpha_j \cdot c_j \right) + \alpha_i (w_i + c_i) \right) \cdot W_{total}, \tag{1.6}$$

$$T = T_{i+1} = \left(\left(\sum_{j=1}^{i-1} \alpha_j \cdot c_j \right) + \alpha_i \cdot c_i + \alpha_{i+1} (w_{i+1} + c_{i+1}) \right) \cdot W_{total}. \tag{1.7}$$

With $K = \frac{T - W_{total} \cdot \left(\sum_{j=1}^{i-1} \alpha_j \cdot c_j \right)}{W_{total}}$, Equations (1.6) and (1.7) can be respectively rewritten as:

$$\alpha_i = \frac{K}{w_i + c_i} \text{ and } \alpha_{i+1} = \frac{K - \alpha_i \cdot c_i}{w_{i+1} + c_{i+1}}.$$

The time needed by communications can be written as:

$$(\alpha_i \cdot c_i + \alpha_{i+1} \cdot c_{i+1}) \cdot W_{total} = \left(\left(\frac{c_i}{w_i + c_i} + \frac{c_{i+1}}{w_{i+1} + c_{i+1}} \right) - \frac{c_i \cdot c_{i+1}}{(w_i + c_i)(w_{i+1} + c_{i+1})} \right) \cdot K \cdot W_{total}.$$

We see that the equation is symmetric, thus that the communication time is completely independent of the order of the two communications.

The total fraction of work is equal to:

$$\alpha_i + \alpha_{i+1} = \left(\frac{1}{w_i + c_i} + \frac{1}{w_{i+1} + c_{i+1}} \right) \cdot K - \frac{K \cdot c_i}{(w_i + c_i)(w_{i+1} + c_{i+1})}.$$

If we exchange P_i and P_{i+1}, then the total fraction of work is equal to:

$$\alpha_i + \alpha_{i+1} = \left(\frac{1}{w_i + c_i} + \frac{1}{w_{i+1} + c_{i+1}} \right) \cdot K - \frac{K \cdot c_{i+1}}{(w_i + c_i)(w_{i+1} + c_{i+1})}.$$

The difference between both solutions is given by

$$\Delta_{i,i+1} = (c_i - c_{i+1}) \frac{K}{(w_i + c_i)(w_{i+1} + c_{i+1})}.$$

Contrarily to the communication time, the fraction of the load done by processors P_i and P_{i+1} depends on the order of the communications. If we serve the fastest-communicating processor first, the fraction of the load processed by P_i and P_{i+1} can be increased without increasing the communication time for other workers. In other terms, in any optimal solution, participating workers should be served by non-decreasing values of c_i. ■

Lemma 1.6. *When processors are numbered in non-decreasing values of the c_i's, the following closed-form formula for the fraction of the total load allocated to processor P_i $(1 \leq i \leq n)$ defines an optimal solution:*

$$\alpha_i = \frac{\frac{1}{c_i+w_i}\prod_{k=1}^{i-1}\left(\frac{w_k}{c_k+w_k}\right)}{\sum_{i=1}^{p}\frac{1}{c_i+w_i}\prod_{k=1}^{i-1}\frac{w_k}{c_k+w_k}}.$$

Proof. Thanks to the previous lemmas, we know that all workers participate in the computation (Lemma 1.3) and have to be served by non-decreasing values of c_i (Lemma 1.5) and that all workers finish simultaneously (Lemma 1.4).

Without loss of generality, we assume that $c_1 \leq c_2 \leq \ldots \leq c_n$. By definition, the completion time of the i-th processor is given by:

$$T_i = T = W_{total}\left(\sum_{k=1}^{i}\alpha_k c_k + \alpha_i w_i\right) = \alpha_i W_{total}\left(w_i + c_i\right) + \sum_{k=1}^{i-1}\left(\alpha_k c_k W_{total}\right). \quad (1.8)$$

Then, an immediate induction solves this triangular system and leads to:

$$\alpha_i = \frac{T}{W_{total}}\frac{1}{c_i + w_i}\prod_{k=1}^{i-1}\left(\frac{w_k}{c_k + w_k}\right). \quad (1.9)$$

To compute the total completion time T, we just need to remember that $\sum_{i=1}^{n}\alpha_i = 1$, hence

$$\frac{T}{W_{total}} = \frac{1}{\sum_{i=1}^{p}\frac{1}{c_i+w_i}\prod_{k=1}^{i-1}\frac{w_k}{c_k+w_k}}.$$

So, we can have the complete formula for the α_i's:

$$\alpha_i = \frac{\frac{1}{c_i+w_i}\prod_{k=1}^{i-1}\left(\frac{w_k}{c_k+w_k}\right)}{\sum_{i=1}^{p}\frac{1}{c_i+w_i}\prod_{k=1}^{i-1}\frac{w_k}{c_k+w_k}}.$$

■

We summarize the previous analysis as:

Theorem 1.2. *For Divisible Loads applications on star-shaped networks, in an optimal solution:*

- *workers are ordered by non-decreasing values of c_i,*
- *all participate in the work,*
- *they all finish simultaneously.*

Closed-form formulas give the fraction of the load allocated to each processor.

We conclude this section with two remarks.

- If the order of the communications cannot be freely chosen, Lemma 1.3 is not always true; for instance, when sending a piece of work to a processor is more expensive than having it processed by the workers served after that processor [47].
- In this section, we considered that the master processor did not participate in the processing. To deal with cases where we can enroll the master in the computation, we can easily add a virtual worker, with the same processing speed as the master, and a communication time equal to 0. Finally, if we also have the freedom to choose the master processor, the simplest method to determine the best choice is to compute the total completion time in all cases: we have only n different cases to check, so we can still determine the best master efficiently (although not as elegantly as for a bus-shaped platform).

1.3 Extensions of the Divisible Load Model

In the previous section, we have shown that the Divisible Load approach enables to solve scheduling problems that are hard to solve under classical models. With a classical approach, we are often limited to solving simpler problems, featuring homogeneous communication links and linear cost functions for computations and communications. In Section 1.2, we have seen how the Divisible Load model could help solve a problem that was already difficult even with homogeneous resources.

Realistic platform models are certainly intractable, and crude ones are not likely to be realistic. A natural question arises: how far can we go with Divisible Load theory? We present below several extensions to the basic framework described so far. In Section 1.3.1, we extend the linear cost model for communications by introducing *latencies*. Next, we distribute chunks to processors in *several* rounds in Section 1.3.2.

In the latter two extensions, we still neglect the cost of *return messages*, assuming that data is only sent from the master to the workers, and that results lead to negligible communications. To conclude this section, we introduce return messages in Section 1.3.3.

1.3.1 Introducing Latencies

In the previous sections, we used a simple linear cost model for communications. The time needed to transmit a data chunk was perfectly proportional to its size. On real-life platforms, there is always some latency to account for. In other words, an affine communication and affine computation model would be more realistic. We have to introduce new notations for these latencies: C_i denotes the communication latency paid by worker P_i for a communication from the master, and W_i denotes the latency corresponding to initializing a computation. If P_i has to process a fraction α_i of the total load, then its communication time is equal to $C_i + c_i\alpha_i W_{total}$ and its computation time is equal to $W_i + w_i\alpha_i W_{total}$.

This variant of the problem is NP-complete [88] and thus much more difficult than the previous one. However, some important results can be shown for any optimal solution:

- Even if communication times are fixed (bandwidths are supposed to be infinite), the problem remains NP-complete, as shown by Yang, Casanova, Drozdowski, Lawenda, and Legrand in [88].
- All participating workers end their work at the same time (Lemma 1.7).
- If the load is large enough, all workers participate in the work and must be served in non-decreasing order of the c_i's (Lemma 1.9).
- An optimal solution can be found using a mixed linear program (Lemma 1.8).

Lemma 1.7. *In any optimal solution, all participating workers have the same completion time.*

Proof. This lemma can be shown using a proof similar to the proof of Lemma 1.4. A detailed proof is given in [13]. ■

The following mixed linear program (1.10) aims at computing an optimal distribution of the load among workers.

$$
\begin{cases}
\text{MINIMIZE } T_f \text{ UNDER THE CONSTRAINTS} \\
\text{(1.10a)} \quad \forall i, 1 \leq i \leq n, \quad \alpha_i \geq 0 \\
\text{(1.10b)} \quad \sum_{i=1}^{n} \alpha_i = 1 \\
\text{(1.10c)} \quad \forall j, 1 \leq j \leq n, , \quad y_j \in \{0,1\} \\
\text{(1.10d)} \quad \forall i, j, 1 \leq i, j \leq n, \quad x_{i,j} \in \{0,1\} \\
\text{(1.10e)} \quad \forall j, 1 \leq i \leq n, \quad \sum_{i=1}^{n} x_{i,j} = y_j \\
\text{(1.10f)} \quad \forall i, 1 \leq i \leq n, \quad \sum_{j=1}^{n} x_{i,j} \leq 1 \\
\text{(1.10g)} \quad \forall j, 1 \leq i \leq n, \quad \alpha_j \leq y_j \\
\text{(1.10h)} \quad \forall i, 1 \leq i \leq n, \quad \sum_{k=1}^{i-1} \sum_{j=1}^{n} x_{k,j}(C_j + \alpha_j c_j W_{total}) \\
\qquad\qquad + \sum_{j=1}^{n} x_{i,j}(C_j + \alpha_j c_j W_{total} + W_j + \alpha_j w_j W_{total}) \leq T_f
\end{cases}
\tag{1.10}
$$

Lemma 1.8. *An optimal solution can be found using the mixed linear program above (with a potentially exponential computation cost).*

Proof. In [14], the authors added the resource selection issue to the original linear program given by Drozdowsky in [41]. To address this issue, they added two notations: y_j, which is a binary variable equal to 1 if, and only if, P_j participates in the work, and $x_{i,j}$, which is a binary variable equal to 1 if, and only if, P_j is chosen for the i-th communication from the master. Equation (1.10e) implies that P_j is involved in exactly y_j communication. Equation (1.10f) states that that at most one worker is used for the i-th communication. Equation (1.10g) ensures that non-participating workers have no work to process. Equation (1.10h) implies that the worker selected for the i-th communication must wait for the previous communications before starting its own communication and then its computation.

This linear program always has a solution, which provides the selected workers and their fraction of the total load in an optimal solution. ■

Lemma 1.9. *In any optimal solution, and if the load is large enough, all workers participate in the work and must be served in non-decreasing order of communication time c_i.*

Proof. We want to determine the total amount of work which can be done in a time T. Let us consider any valid solution to this problem. The set of the k active workers is denoted $\mathcal{S} = \{P_{\sigma(1)}, \ldots, P_{\sigma(k)}\}$, where σ is a one-to-one mapping from $[1 \ldots k]$ to $[1 \ldots n]$ and represents the order of communications. Let n_{TASK} denote the maximum number of processed units of load using this set of workers in this order.

- Consider the following instance of our problem, with k workers $P'_{\sigma(1)}$, …, $P'_{\sigma(k)}$, such that $\forall i \in \{1, \ldots, k\}, C'_i = 0, W'_i = 0, c'_i = c_i, w'_i = w_i$ (in fact, we are just ignoring all latencies). The total number of work units n'_{TASK} which can be executed on this platform in the time T is greater than the number n_{TASK} of tasks processed by the original platform:

$$n_{TASK} \leq n'_{TASK}.$$

Using Theorem 1.2, we know that n'_{TASK} is given by a formula of the following form:

$$n'_{TASK} = f(\mathcal{S}, \sigma) \cdot T.$$

The main point is that n'_{TASK} is proportional to T.

- Now we will determine the number of works units n''_{TASK}, which could be done in a time $T'' = T - \sum_{i \in \mathcal{S}}(C_i + W_i)$. n''_{TASK} is clearly smaller than n_{TASK} since it consists in adding all latencies before the beginning of the work:

$$n''_{TASK} \leq n_{TASK}.$$

The previous equality still stands:

$$n''_{TASK} = f(\mathcal{S}, \sigma) \left(T - \sum_{i \in \mathcal{S}} (C_i + W_i) \right).$$

We have $n''_{TASK} \leq n_{TASK} \leq n'_{TASK}$ and then

$$f(\mathcal{S}, \sigma) \left(1 - \frac{\sum_{i \in \mathcal{S}} (C_i + W_i)}{T} \right) \leq \frac{n_{TASK}}{T} \leq f(\mathcal{S}, \sigma).$$

Therefore, when T becomes arbitrarily large, the throughput of the platform becomes close to the theoretical model without any latency. Thus, when T is sufficiently large, in any optimal solution, all workers participate in the work, and chunks should be sent on the ordering of non-decreasing communication times c_i.

Without any loss of generality, we can assume that $c_1 \leq \ldots \leq c_p$ and then the following linear system returns an asymptotically optimal solution:

$$\begin{cases} \text{MINIMIZE } T \text{ UNDER THE CONSTRAINTS} \\ \text{(1.11a)} \quad \displaystyle\sum_{i=1}^{n} \alpha_i = 1 \\ \text{(1.11b)} \quad \forall i \in \{1, \ldots, n\}, \quad \displaystyle\sum_{k=1}^{i} (C_k + c_k \alpha_k W_{total}) + W_i + w_i \alpha_i W_{total} = T \end{cases} \tag{1.11}$$

Moreover, when T is sufficiently large, this solution is optimal when all c_i are different, but determining the best way to break ties among workers with the same communication speed remains an open question.

■

1.3.2 Multi-Round Strategies

So far, we only used models with a strict one-port communication scheme, and data were transmitted to workers in a single message. Therefore, each worker had to wait while previous one were communicating with the master before it could begin receiving and processing its own data. This waiting time can lead to a poor utilization of the platform. A natural solution to quickly distribute some work to each processor is to distribute data in multiple rounds: while the first processor is computing its first task, we can distribute data to other workers, and then resume the distribution to the first processor, and so on. By this way, we hope to overlap communications with computations, thereby increasing platform throughput (see Figure 1.11). This idea seems promising, but we have two new questions to answer:

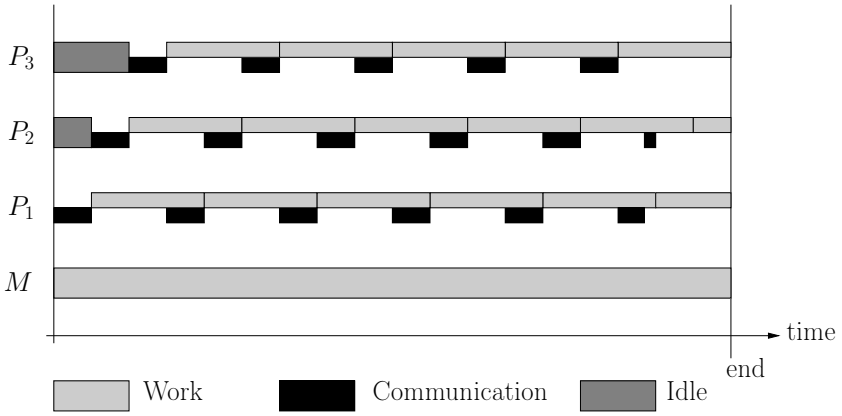


Figure 1.11: Multi-round execution over a bus-shaped platform.

1. how many rounds should we use to distribute the whole load?

2. which size should we allocate to each round?

If we follow the lesson learnt using a single-round distribution, we should try to solve the problem without latencies first. However, it turns out that the linear model is not interesting in a multi-round framework: the optimal solution has an infinite number of rounds of size zero, as stated by Theorem 1.3.

Theorem 1.3. *Let us consider any homogeneous bus-shaped master-worker platform, following a linear cost model for both communications and computations. Then any optimal multi-round schedule requires an infinite number of rounds.*

This result was widely accepted but was not formally shown. We give a complete demonstration in [A1].

On the contrary, when latencies are added to the model, they prevent solutions using such an infinite number of rounds. However, the problem becomes NP-complete. In the following, we first assume that a maximum number of rounds is given and we explain how to obtain optimal solutions, before presenting asymptotically optimal solutions when there is no bound on the number of rounds. Finally, we assess the gain of multi-round strategies with respect to one-round solutions. In the next section, we abandon the linear cost model leading to this infinite number of rounds and we introduce latencies.

Bus-Shaped Network and Homogeneous Processors, Fixed Number of Rounds

The simplest case to explore is a bus-shaped network of homogeneous processors, i.e., with homogeneous communication links, and when the number of rounds to use is given to the algorithm.

Intuitively, rounds have to be small to allow a fast start of computations, but also have to be large to amortize the cost of latencies. These two contradictory objectives can be merged by using small rounds at the beginning, and then increasing them progressively to amortize paid latencies.

The first work on multi-round strategies was done by Bharadwaj et al. using a linear cost model for both communications and computations [27]. This was followed by Yang and Casanova in [87], who used affine models instead of linear ones. In the following, we present their solution.

Since we only consider homogeneous platforms, we have for any worker i (with $1 \leq i \leq n$) $w_i = w$, $W_i = W$, $c_i = c$, and $C_i = C$. Moreover, R denotes the computation-communication ratio ($R = w/c$) and γ_j denotes the time to compute the chunk j excluding the computation latency: $\gamma_j = \alpha_j \cdot w \cdot W_{total}$. We assume that we distribute the whole load in M rounds of n chunks. For technical reasons, chunks are numbered in the reverse order, from $Mn-1$ (the first one) to 0 (the last one).

Using these notations, we can write the recursion on the γ_j series:

$$\forall j \geq n, W + \gamma_j = \frac{1}{R}(\gamma_{j-1} + \gamma_{j-2} + \ldots + \gamma_{j-n}) + n \cdot C, \tag{1.12}$$

$$\forall 0 \leq j < n, W + \gamma_j = \frac{1}{R}(\gamma_{j-1} + \gamma_{j-2} + \ldots + \gamma_{j-n}) + j \cdot C + \gamma_0, \tag{1.13}$$

$$\forall j < 0, \gamma_j = 0. \tag{1.14}$$

Equation (1.12) expresses that a worker j must receive enough data to compute during exactly the time needed for the next n chunks to be communicated, ensuring no idle time on the communication bus. This equation is of course not true for the last n chunks. Equation (1.13) states that all workers have to finish their work at the same time. Finally, Equation (1.14) ensures the correctness of the two previous equations by setting out-of-range terms to 0.

This recursion corresponds to an infinite series in the γ_j, of which the first nM values give the solution to our problem. Using generating functions, we can solve this recursion. Let $\mathcal{G}(x) = \sum_{j=0}^{\infty} \gamma_j x^j$ be the generating function for the series. Using Equations (1.12) and (1.13), the value of $\mathcal{G}(x)$ can be expressed as (see [87]):

$$\mathcal{G}(x) = \frac{(\gamma_0 - n \cdot C)(1 - x^n) + (n \cdot C - W) + C \cdot \left(\frac{x(1-x^{n-1})}{1-x} - (n-1)x^n\right)}{(1-x) - x(1-x^n)/R}.$$

The rational expansion method [50] gives the roots of the polynomial denominator and then the correct values of the γ_j's, and finally, the values of the α_j's. The value of the first term γ_0 is given by the equation $\sum_{j=0}^{Mn-1} \gamma_j = W_{total} \cdot w$.

Bus-Shaped Network, Computing the Number of Rounds

In the previous section, we assumed that the number of rounds was given to the algorithm, thus we avoided one of the two issues of multi-round algorithms. Now, we suppose that the number of chunks has to be computed by the scheduling algorithm as well as their respective sizes. As we said before, we have to find a good compromise between a small number of chunks, to reduce the overall cost of latencies, and a large one, to ensure a good overlap of communications by computations.

In fact, finding the optimal number of rounds for such algorithms and affine cost models is still an open question. Nonetheless, Yang, van der Raadt, and Casanova proposed the Uniform Multi-Round (UMR) algorithm [89]. This algorithm is valid in the homogeneous case as well as in the heterogeneous case, but we will only look at the homogeneous one for simplicity reasons. To simplify the problem, UMR assume that all chunks sent to workers during the same round

have the same size. This constraint can limit the overlap of communications by computations, but it allows to find an optimal number of rounds.

In this section, α_j denotes the fraction of the load sent to any worker during the j-th round, and M denotes the total number of rounds. Then there are n chunks of size α_j, for a total of $n \cdot M$ chunks. The constraint of uniform sizes for chunks of the same round is not used for the last round, allowing the workers to finish simultaneously. To ensure a good utilization of the communication link, the authors force the master to finish sending work for round $j+1$ to all workers when worker p finishes computing for round j. This condition can be written as:

$$W + \alpha_j \cdot w \cdot W_{total} = p \cdot (C + \alpha_{j+1} \cdot c \cdot W_{total}),$$

which leads to:

$$\alpha_j = \left(\frac{c}{n \cdot w}\right)^j (\alpha_0 - \gamma) + \gamma, \tag{1.15}$$

where $\gamma = \frac{1}{w - n \cdot c} \cdot (n \cdot C - W)$. The case $w = n \cdot c$ is simpler and detailed in the original paper [89].

With this simple formula, we can give the makespan $\mathcal{M}$ of the complete schedule, which is the sum of the time needed by the worker n to process its data, the total latency of computations and the time needed to send all the chunks during the first round (the $\frac{1}{2}$ factor comes from the non-uniform sizes of the last round, since all workers finish simultaneously):

$$\mathcal{M}(M, \alpha_0) = \frac{W_{total}}{n} + M \cdot W + \frac{1}{2} \cdot n \cdot (C + c \cdot \alpha_0). \tag{1.16}$$

The complete schedule needs to process the entire load, which can be written as:

$$\mathcal{G}(M, \alpha_0) = \sum_{j=0}^{M-1} p \cdot \alpha_j = W_{total}. \tag{1.17}$$

Using these equations, the problem can be expressed as minimizing $\mathcal{M}(M, \alpha_0)$ subject to $\mathcal{G}(M, \alpha_0)$. The Lagrange Multiplier method [24] leads to a single equation, which cannot be solved analytically but only numerically. Several simulations showed that uniform chunks can reduce performance, when compared to the multi-round algorithm of the previous section, when latencies are small; but they lead to better results when latencies are large. Moreover, the UMR algorithm can be used on heterogeneous platforms, contrarily to the previous multi-round algorithm.

Asymptotically optimal algorithm. Finding an optimal algorithm that distributes data to workers in several rounds, is still an open question. Nevertheless, it is possible to design asymptotically optimal algorithms. An algorithm is asymptotically optimal if the ratio of its makespan obtained with a load W_{total} over the optimal makespan with this same load tends to 1 as W_{total} tends to infinity. This approach is coherent with the fact that the Divisible Load model already is an approximation well suited to large workloads.

Theorem 1.4. *Consider a star-shaped platform with arbitrary values of computation and communication speeds and latencies, allowing the overlap of communications by computations. There exists an asymptotically optimal periodic multi-round algorithm.*

Proof. The main idea is to look for a periodic schedule: the makespan T is divided into k periods of duration T_p. The initialization and the end of the schedule are sacrificed, but the large

duration of the whole schedule amortizes this sacrifice. We still have to find a good compromise between small and large periods. It turns out that choosing a period length proportional to the square-root of the optimal makespan T^* is a good trade-off. The other problem is to choose the participating workers. This was solved by Beaumont et al. using linear programming [13]. If $\mathcal{I} \subseteq \{1, \ldots, p\}$ denotes the selected workers, we can write that communication and computation resources are not exceeded during a period of duration T_p:

$$\sum_{i \in \mathcal{I}} (C_i + \alpha_i \cdot c_i W_{total}) \leq T_p,$$

$$\forall i \in \mathcal{I}, W_i + \alpha_i \cdot w_i \cdot W_{total} \leq T_p.$$

We aim to maximize the average throughput $\rho = \sum_{i \in \mathcal{I}} \frac{\alpha_i \cdot W_{total}}{T_p}$, where $\frac{\alpha_i \cdot W_{total}}{T_p}$ is the average number of load units processed by P_i in one time unit, under the following linear constraints:

$$\begin{cases} \forall i \in \mathcal{I}, & \frac{\alpha_i \cdot W_{total}}{T_p} w_i \leq 1 - \frac{W_i}{T_p} & \text{(overlap)}, \\ & \sum_{i \in \mathcal{I}} \frac{\alpha_i \cdot W_{total}}{T_p} c_i \leq 1 - \frac{\sum_{i \in \mathcal{I}} C_i}{T_p} & \text{(1-port model)} \end{cases}$$

This set of constraints can be replaced by the following one, stronger but easier to solve:

$$\begin{cases} \forall i \in \{1, \ldots, p\}, & \frac{\alpha_i \cdot W_{total}}{T_p} w_i \leq 1 - \frac{\sum_{i=1}^{p} C_i + W_i}{T_p} & \text{(overlap)}, \\ & \sum_{i=1}^{p} \frac{\alpha_i \cdot W_{total}}{T_p} c_i \leq 1 - \frac{\sum_{i=1}^{p} C_i + W_i}{T_p} & \text{(1-port model)} \end{cases} \tag{1.18}$$

Without any loss of generality, assume that $c_1 \leq c_2 \leq \ldots \leq c_p$ and let q be the largest index, such that $\sum_{i=1}^{q} \frac{c_i}{w_i} \leq 1$. Let ε be equal to $1 - \sum_{i=1}^{q} \frac{c_i}{w_i}$ if $q < p$, and to 0 otherwise. Then the optimal throughput for system (1.18) is realized with

$$\begin{aligned} \forall 1 \leq i \leq q, \quad \frac{\alpha_i \cdot W_{total}}{T_p} &= \frac{1}{c_i} \left(1 - \frac{\sum_{i=1}^{p} C_i + W_i}{T_p}\right) \\ \frac{\alpha_{q+1} \cdot W_{total}}{T_p} &= \left(1 - \frac{1}{T_p} \sum_{i=1}^{p} (C_i + W_i)\right) \left(\frac{\varepsilon}{c_{q+1}}\right) \\ \forall q+2 \leq i \leq n \quad \alpha_i &= 0 \end{aligned}$$

and the throughput is equal to

$$\rho = \sum_{i=1}^{p} \frac{\alpha_i \cdot W_{total}}{T_p} = \left(1 - \frac{\sum_{i=1}^{p} C_i + W_i}{T_p}\right) \rho_{\text{opt}} \text{ with } \rho_{\text{opt}} = \sum_{i=1}^{q} \frac{1}{w_i} + \frac{\varepsilon}{c_{q+1}}.$$

To prove the asymptotic optimality of this algorithm, we need an upper bound on the optimal throughput ρ^*. This upper bound can be obtained by removing all latencies (i.e., $C_i = 0$ and $W_i = 0$ for any worker i). By definition, we have $\rho^* \leq \rho_{\text{opt}}$. If we call T^* the optimal time needed to process B load units, then we have

$$T^* \geq \frac{B}{\rho^*} \geq \frac{B}{\rho_{opt}}.$$

Let T denote the time needed by the proposed algorithm to compute the same workload B. The first period is dedicated to communications and is lost for processing, so $k = \left\lceil \frac{B}{\rho \cdot T_p} \right\rceil + 1$ periods are required for the whole computation.

We have $T = k \cdot T_p$, therefore:

$$T \leq \frac{B}{\rho} + 2 \cdot T_p \leq \frac{B}{\rho_{opt}} \left(\frac{1}{1 - \sum_{i=1}^{p} \frac{C_i + W_i}{T_p}} \right) + 2 \cdot T_p,$$

and, if $T_p \geq 2 \cdot \sum_{i=1}^{p} C_i + W_i$,

$$T \leq \frac{B}{\rho_{\text{opt}}} + 2 \cdot \frac{B}{\rho_{\text{opt}}} \sum_{i=1}^{p} \frac{C_i + W_i}{T_p} + 2 \cdot T_p$$

and if T_p is equal to $\sqrt{\frac{B}{\rho_{\text{opt}}}}$,

$$\frac{T}{T^*} \leq 1 + 2 \left(\sum_{i=1}^{p} (C_i + W_i) + 1 \right) \frac{1}{\sqrt{T^*}} = 1 + O\left(\frac{1}{\sqrt{T^*}} \right).$$

That suffices to show the asymptotic optimality of the proposed algorithm. ■

Maximum benefit of multi-round algorithms. Using multi-round algorithms brings new difficulties to an already difficult problem. It is worth to assess how much such algorithms can improve the solution. We answered this question with the following result:

Theorem 1.5. *Consider any star-shaped master-worker platform where communication cost and computation cost each follows either a linear or an affine model. Any multi-round schedule cannot improve an optimal single-round schedule by a factor larger than* 2.

Proof. Let $\mathcal{S}$ be any optimal multi-round schedule, using a finite number K of rounds. We have n workers in our platform, and the master allocates a fraction $\alpha_i(k)$ to worker i ($1 \leq i \leq n$) during the k-th round. Let T denote the total completion time obtained by $\mathcal{S}$. From $\mathcal{S}$ we build a new schedule $\mathcal{S}'$ which sends in a single message a fraction $\sum_{k=1}^{K} \alpha_i(k)$ to worker i (the messages are sent in an arbitrary order.) The master does not spend more time communicating under $\mathcal{S}'$ than under $\mathcal{S}$. Therefore, no later than time T all workers will have finished to receive their work. No worker will spend more time processing its fraction of the load under $\mathcal{S}'$ than under $\mathcal{S}$ (the loads have same sizes). Therefore, none of them will spend more than T time-units to process its load under $\mathcal{S}'$. Therefore, the makespan of the single round schedule $\mathcal{S}'$ is no greater than $2T$.

■

1.3.3 Return Messages

In previous sections, we assumed that computations required some input data but produced a negligible output, so we did not take its transmission back to the master into account. This assumption could be unduly restrictive for those computations producing large outputs, such as cryptographic keys. In this section, we incorporate return messages into the story. Which properties are still valid?

The problem has been studied in [2], its precursor [69] and in [1], showing that taking these return messages into account has a dramatic impact on the design of algorithms. In the general case, there is no correlation between input and output sizes, but we simplify the problem by

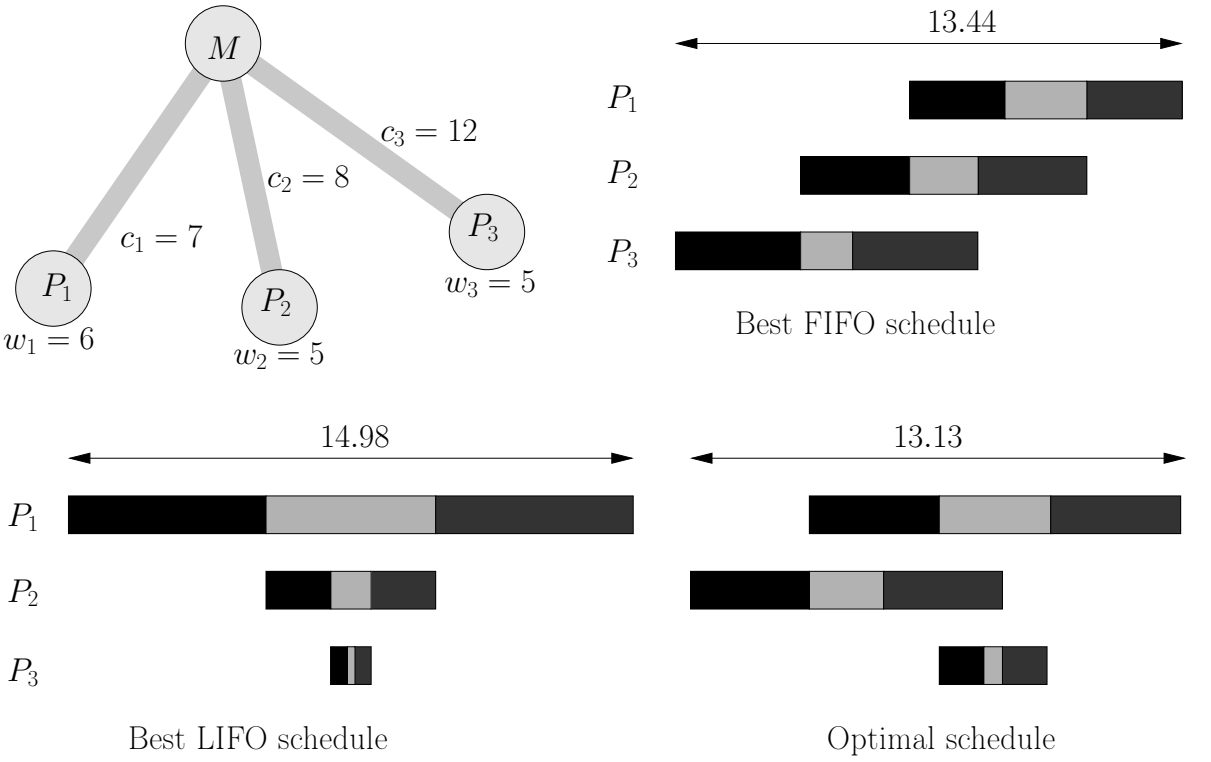


Figure 1.12: Optimal order can be neither FIFO nor LIFO.

assuming the same size for input and output messages. In other words, if M needs $c_i\alpha_i W_{total}$ time units to send the input to worker P_i, P_i needs the same time to send the result back to M after having completed its computation. The communication medium is supposed to be bi-directional (as most of network cards are now full-duplex), so the master M can simultaneously send and receive data.

In our first framework with linear cost models and distribution in a single round, all workers participated in the work and we were able to find an optimal order to distribute data. If we allow return messages, we have two new issues: the order of return messages could be different from the distribution order and several workers could remain idle during the whole computation. Two simple ordering strategies are the FIFO strategy (return messages are sent in the same order as the input messages) and the LIFO strategy (return messages are sent in reverse order). In fact, several examples are exhibited in [20], where the optimal order for return messages is neither FIFO or LIFO, as illustrated by Figure 1.12, or where the optimal makespan is reached with some idle processors, as illustrated by Figure 1.13. The best FIFO and LIFO distribution are easy to compute, since all processors are involved in the work, they are served in non-decreasing values of communication times and do not remain idle between the initial message and the return message [20]. Furthermore, all FIFO schedules are optimal in the case of a bus-shaped platform [2].

Moreover, the distribution of data in a single round induces long waiting times, and a multi-round distribution could really improve this drawback. Regrettably, any optimal multi-round distribution for the linear cost model uses an infinite number of rounds. Thus, affine cost models are required to have realistic solutions, and the problem then becomes very hard to solve or even

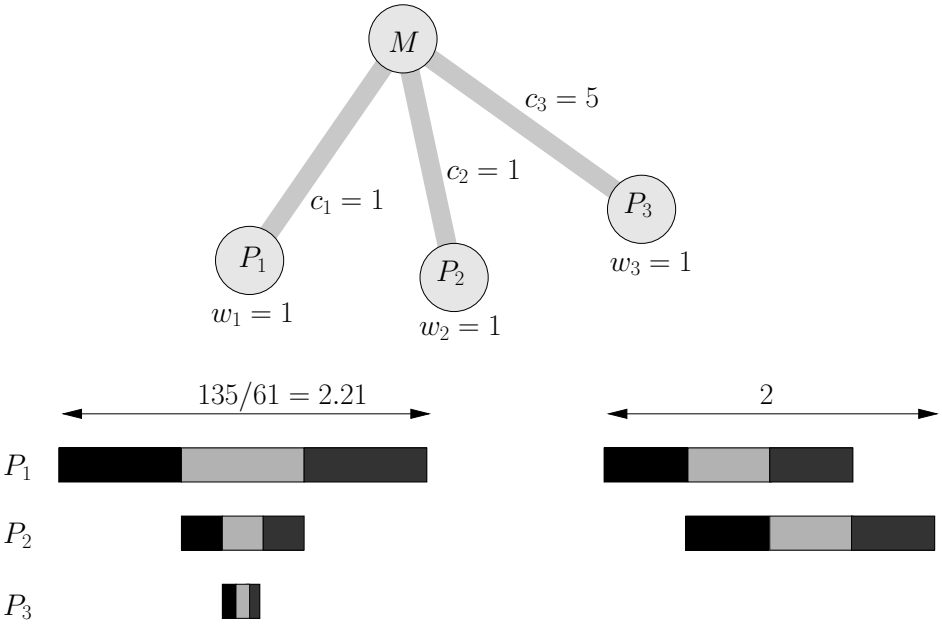


Figure 1.13: Example with an idle worker.

to approximate.

1.4 Conclusion

In this chapter, we have dealt with the Divisible Load model, a simple and useful relaxation to many scheduling problems. A general applicative example, the execution of a distributed computation made of independent tasks on a star-shaped platform, has been used as a guideline through this chapter. Without any relaxation, this example is a tractable problem, but the known solution to this problem is only partial and has a large computational complexity. Moreover, the linear cost function used for communications and computations, and the homogeneous communication model, limit the practical significance of this approach. We showed how to use the Divisible Load theory to simplify the problem and then solve it completely. Because we have simplified the problem with the Divisible Load relaxation, we can afford to use a more complicated model: we can deal with heterogeneous communication links and/or we can include latencies in communications and computations. This new model is more realistic, but still tractable thanks to the Divisible Load approach. However, it also has its own limits! Indeed, we have seen that problem complexity quickly increases as soon as we add latencies.

Chapter 2

Scheduling divisible loads on a chain of processors

2.1 Introduction

Several papers in the Divisible Load Theory field consider master-worker platforms [28, 47, 14]. However, in communication-bound situations, a linear network of processors can lead to better performance: on such a platform, several communications can take place simultaneously, thereby enabling a pipelined approach. Recently, Min, Veeravalli, and Barlas have proposed strategies to minimize the overall execution time of one or several divisible loads on a heterogeneous linear processor network [84, 85]. Initially, the authors targeted single-installment strategies, that is strategies under which a processor receives all its share of a load in a single communication. But for cases where their approach failed to design single-installment strategies, they also considered multi-installment solutions.

In this chapter, we define the framework and some notations in Section 2.2. Next we show with a very simple example (Section 2.3) that the approach proposed in [85] does not always produce a solution and that, when it does, the solution is often suboptimal. The fundamental flaw of the approach of [85] is that the authors are optimizing the scheduling load by load, instead of attempting a global optimization. The load by load approach is suboptimal and unduly over-constrains the problem. On the contrary, we show (Section 2.4) how to find an optimal scheduling for any instance, once the number of installments per load is given. In particular, our approach always finds the optimal solution in the single-installment case. We also formally state (Section 2.5) that under a linear cost model for communication and communication, as in [84, 85], an optimal schedule has an infinite number of installments. Such a cost model can therefore not be used to design practical multi-installment strategies. Finally, in Section 2.6, we report the simulations that we performed in order to assess the actual efficiency of the different approaches. We now start by introducing the framework.

2.2 Problem and notations

We summarize here the framework of [84, 85]. The target architecture is a linear chain of n processors (P_1, P_2, ..., P_n). Processor P_i is connected to processor P_{i+1} by the communication link l_i (see Figure 2.1). The target application is composed of m loads, which are *divisible*, meaning that each load can be split into an arbitrary number of chunks of any size, and these chunks can be processed independently. All the loads are initially available on processor P_1,

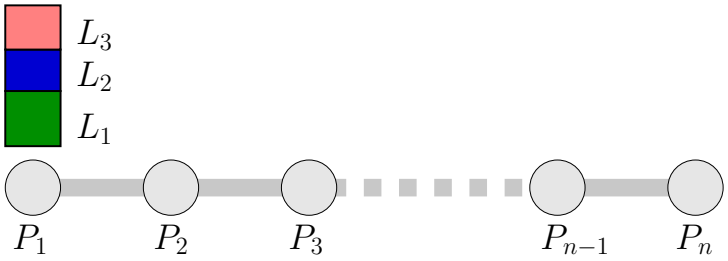


Figure 2.1: Linear network, with n processors and $n-1$ links.

which processes a fraction of them and delegates (sends) the remaining fraction to P_2. In turn, P_2 executes part of the load that it receives from P_1 and sends the rest to P_3, and so on along the processor chain. Communications can be overlapped with (independent) computations, but a given processor can be active in at most a single communication at any time-step: sends and receives are serialized (this is the strict *one-port* model).

Since the last processor P_n cannot start computing before having received its first message, it is useful for P_1 to distribute the loads in several installments: the idle time of remote processors in the chain will be reduced due to the fact that communications are smaller in the first steps of the overall execution.

We deal with the general case in which the k-th load is distributed in Q_k installments of different sizes. For the j-th installment of load k, processor P_i takes a fraction $\gamma_j^k(i)$, and sends the remaining part to the next processor while processing its own fraction (that is, processor P_i sends a volume of data equal to $\sum_{u=i+1}^{n} \gamma_j^k(u)$ to processor P_{i+1}).

In the framework of [84, 85], loads have different characteristics. Every load k (with $1 \leq k \leq m$) is defined by a volume of data $V_{comm}(k)$ and a quantity of computation $V_{comp}(k)$. Moreover, processors and links are not identical either. We let s_i be the speed of processor P_i ($1 \leq i \leq n$), and bw_i be the bandwidth used by P_i to send a load to P_{i+1} (over link l_i, $1 \leq i \leq n-1$). Note that we assume a linear model for computations and communications, as in the original articles, and as is often the case in divisible load literature [28, 48, 68].

For the j-th installment of the k-th load, let $Comm_{i,k,j}^{start}$ denote the starting time of the communication between P_i and P_{i+1}, and let $Comm_{i,k,j}^{end}$ denote its completion time; similarly, $Comp_{i,k,j}^{start}$ denotes the start time of the computation on P_i for this installment, and $Comp_{i,k,j}^{end}$ denotes its completion time. The objective function is to minimize the *makespan*, i.e., the time at which all loads are computed.

2.3 An illustrative example

2.3.1 Presentation

To show the limitations of [84, 85], we deal with a simple revealing example. We use 2 identical processors P_1 and P_2 with $s_1 = s_2 = 1/\lambda$, and $bw_1 = 1$. We consider $m = 2$ identical divisible loads to process, with $V_{comm}(1) = V_{comm}(2) = 1$ and $V_{comp}(1) = V_{comp}(2) = 1$. Note that when λ is large, communications become negligible and each processor is expected to process around half of both loads. But when λ is close to 0, communications are very important, and the solution is not obvious.

We first consider a simple schedule which uses a single installment for each load. Processor P_1 computes a fraction $\gamma_1^1(1) = \frac{2\lambda^2+1}{2\lambda^2+2\lambda+1}$ of the first load, and a fraction $\gamma_1^1(2) = \frac{2\lambda+1}{2\lambda^2+2\lambda+1}$ of the second load. Then the second processor computes a fraction $\gamma_2^1(1) = \frac{2\lambda}{2\lambda^2+2\lambda+1}$ of the first load, and a fraction $\gamma_2^1(2) = \frac{2\lambda^2}{2\lambda^2+2\lambda+1}$ of the second load. The makespan achieved by this schedule is equal to $makespan_1 = \frac{2\lambda(\lambda^2+\lambda+1)}{2\lambda^2+2\lambda+1}$.

In order to further simplify equations, we write α instead of $\gamma_2^1(1)$ (i.e., α is the fraction of the first load sent from the first processor to the second one), and β instead of $\gamma_2^2(1)$ (similarly, β is the fraction of the second load sent to the second processor).

We used simpler notations than the ones used in [85]. However, as we want to explicit the solutions proposed by [85] for our example, we need to use the original notations to enable the reader to double-check our statements. The necessary notations from [85] are recalled in Table 2.1.

T_{cp}^k	Time taken by the standard processor ($w = 1$) to compute the load L_k.
T_{cm}^k	Time taken by the standard link ($z = 1$) to communicate the load L_k.
L_k	Size of the k-th load, where $1 \leq k \leq m$.
$L_{l,k}$	Portion of the load L_k assigned to the l-th installment for processing.
$\alpha_{k,i}^{(l)}$	The fraction of the total load $L_{l,k}$ to P_i, where $0 \leq \alpha_{k,i}^{(l)} \leq 1, \quad \forall i = 1, \ldots, n$ and $\sum_{i=1}^{n} \alpha_{k,i}^{(l)} = 1$.
$t_{l,k}$	The time instant at which is initiated the first communication for the l-th installment of load L_k ($L_{l,k}$).
$C_{l,k}$	The total communication time of the l-th installment of load L_k when $L_{l,k} = 1$; $C_{l,k} = \frac{T_{cm}^k}{L_k} \sum_{p=1}^{n-1} \left(\frac{\left(1-\sum_{j=1}^{p} \alpha_{k,j}^{(l)}\right)}{bw_p} \right)$.
$E_{l,k}$	The total processing time of P_n for the l-th installment of load L_k when $L_{l,k} = 1$; $E_{l,k} = \frac{\alpha_{k,n}^{(l)}}{s_n} T_{cp}^k \frac{1}{L_k}$.
$T(l,k)$	The *finish time* of the l-th installment of load L_k; it is defined as the time instant at which the processing of the l-th installment of load L_k ends.
$T(k)$	The *finish time* of the load L_k; it is defined as the time instant at which the processing of the k-th load ends, i.e., $T(k) = T(Q_k)$ where Q_k is the total number of installments required to finish processing load L_k. $T(m)$ is the finish time of the entire set of loads resident in P_1.

Table 2.1: Summary of the notations of [85] used in this chapter.

In the solution of [85], both P_1 and P_2 have to complete the first load at the same time, and the same holds true for the second load. The transmission for the first load will take α time units, and the one for the second load β time units. Since P_1 (respectively P_2) will process the first load during $\lambda(1-\alpha)$ (respectively $\lambda\alpha$) time units and the second load during $\lambda(1-\beta)$ (respectively $\lambda\beta$) time units, we can write the following equations:

$$\begin{aligned} \lambda(1-\alpha) &= \alpha + \lambda\alpha \qquad (2.1) \\ \lambda(1-\alpha) + \lambda(1-\beta) &= (\alpha + \max(\beta, \lambda\alpha)) + \lambda\beta \end{aligned}$$

There are two cases to discuss:

1. $\max(\beta, \lambda\alpha) = \lambda\alpha$: the solution uses a single installment to distribute the entire load.
2. $\max(\beta, \lambda\alpha) = \beta$: in this case, the load is distributed using several installments.

2.3.2 Solution of [85], one-installment

We are in the one-installment case when $L_2 C_{1,2} \leq T(1) - t_{1,2}$, i.e., $\beta \leq \lambda(1-\alpha) - \alpha$ (Equation (5) in [85], where $L_2 = 1$, $C_{1,2} = \beta$, $T(1) = \lambda(1-\alpha)$ and $t_{1,2} = \alpha$). The values of α and β are given by:

$$\alpha = \frac{\lambda}{2\lambda + 1} \qquad \text{and} \qquad \beta = \frac{1}{2}.$$

This case is true for $\lambda\alpha \geq \beta$, i.e., $\frac{\lambda^2}{2\lambda+1} \geq \frac{1}{2} \Leftrightarrow \lambda \geq \frac{1+\sqrt{3}}{2} \approx 1.366$.

In this case, the makespan is equal to:

$$makespan_2 = \lambda(1-\alpha) + \lambda(1-\beta) = \frac{\lambda(4\lambda+3)}{2(2\lambda+1)}.$$

Comparing both makespans, we have:

$$makespan_2 - makespan_1 = \frac{\lambda\left(2\lambda^2 - 2\lambda - 1\right)}{8\lambda^3 + 12\lambda^2 + 8\lambda + 2}.$$

For all $\lambda \geq \frac{\sqrt{3}+1}{2} \approx 1.366$, our solution is better than theirs, since:

$$\frac{1}{4} \geq makespan_2 - makespan_1 \geq 0.$$

Furthermore, the solution of [85] is strictly suboptimal for any $\lambda > \frac{\sqrt{3}+1}{2}$. Intuitively, the solution of [85] is worse than the schedule of Section 2.3.1 because it aims at locally optimizing the makespan for the first load, and then optimizing the makespan for the second one, instead of directly searching for a global optimum. A visual representation of this case is given in Figure 2.2 for $\lambda = 2$.

2.3.3 Solution of [85], multi-installment

In the second case, we have $\max(\beta, \lambda\alpha) = \beta$. P_1 does not have enough time to completely send the second load to P_2 before the end of the computation of the first load on both processors. The way to proceed in [85] is to send the second load using a multi-installment strategy.

By using Equation (2.1), we can compute the value of α:

$$\alpha = \frac{\lambda}{2\lambda + 1}.$$

Then we have $T(1) = (1-\alpha)\lambda = \frac{\lambda+1}{2\lambda+1}\lambda$ and $t_{1,2} = \alpha = \frac{\lambda}{2\lambda+1}$, i.e., the communication for the second request begins as soon as possible.

We know from Equation (1) of [85] that $\alpha^l_{2,1} = \alpha^l_{2,2}$, and by definition of the α's, $\alpha^l_{2,1} + \alpha^l_{2,2} = 1$, so we have $\alpha^l_{2,i} = \frac{1}{2}$. We also have $C_{1,2} = 1 - \alpha^l_{2,1} = \frac{1}{2}$, $E_{1,2} = \frac{\lambda}{2}$, $Y^{(1)}_{1,2} = 0$, $X^{(1)}_{1,2} = \frac{1}{2}$, $H = H(1) = \frac{X^{(1)}_{1,2} C_{1,2}}{C_{1,2}} = \frac{1}{2}$, $B = C_{1,2} + E_{1,2} - H = \frac{\lambda}{2}$.

We will denote by $\beta_1, \ldots, \beta_k$ the sizes of the different installments processed on each processor (then we have $L_{l,2} = 2\beta_l$).

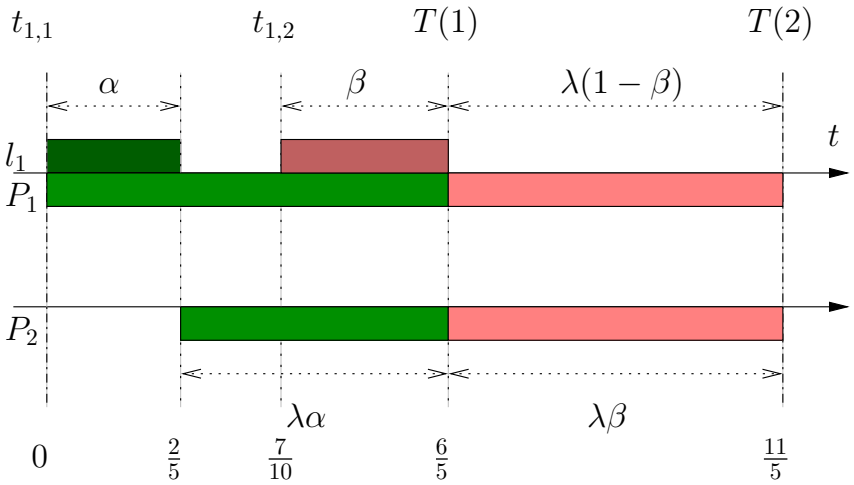


Figure 2.2: The schedule of [85] for $\lambda = 2$, with $\alpha = \gamma_2^1(1)$ and $\beta = \gamma_2^1(2)$.

Since the second processor is not left idle, and since the size of the first installment is such that the communication ends when P_2 completes the computation of the first load, we have $\beta_1 = T(1) - t_{1,2} = \lambda\alpha$ (see Equation (27) in [85], in which we have $C_{1,2} = \frac{1}{2}$).

By the same way, we have $\beta_2 = \lambda\beta_1$, $\beta_3 = \lambda\beta_2$, and so on (see Equation (38) in [85], we recall that $B = \frac{\lambda}{2}$, and $C_{1,2} = \frac{1}{2}$):

$$\beta_l = \lambda^l \alpha.$$

Each processor computes the same fraction of the second load. If we have Q installments, the total processed portion of the second load is upper bounded as follows (when $\lambda \neq 1$):

$$\begin{aligned}\sum_{l=1}^{Q} (2\beta_l) &\leq 2\sum_{l=1}^{Q} \left(\alpha\lambda^l\right) \\ &= 2\frac{\lambda}{2\lambda+1}\lambda\frac{\lambda^Q - 1}{\lambda - 1} \\ &= \frac{2\left(\lambda^Q - 1\right)\lambda^2}{2\lambda^2 - \lambda - 1}.\end{aligned}$$

If $\lambda = 1$, we have $Q = 2$ and:

$$\sum_{l=1}^{Q} (2\beta_l) \leq \frac{2\lambda^2 Q}{2\lambda+1}.$$

We have three sub-cases to discuss:

1. $0 < \lambda < \frac{\sqrt{17}+1}{8} \approx 0.64$: Since $\lambda < 1$, we can write for any nonnegative integer Q:

$$\sum_{l=1}^{Q} (2\beta_l) < \sum_{l=1}^{\infty} (2\beta_l) = \frac{2\lambda^2}{(1-\lambda)(2\lambda+1)}.$$

We have $\frac{2\lambda^2}{(1-\lambda)(2\lambda+1)} < 1$ for all $\lambda < \frac{\sqrt{17}+1}{8}$. So, even in the case of an infinite number of installments, the second load will not be completely processed. In other words, no solution

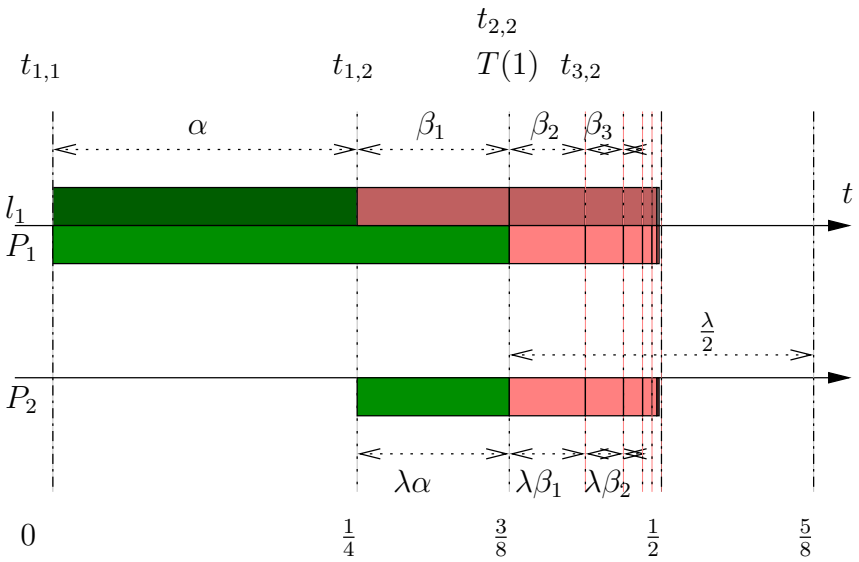


Figure 2.3: The example with $\lambda = \frac{1}{2}$, $\alpha = \gamma_2^1(1)$ and $\beta_l = \gamma_2^l(2)$.

is found in [85] for this case. A visual representation of this case is given in Figure 2.3 with $\lambda = 0.5$.

2. $\lambda = \frac{\sqrt{17}+1}{8}$: We have $\frac{2\lambda^2}{(1-\lambda)(2\lambda+1)} = 1$, so an infinite number of installments is required to completely process the second load. Again, this solution is obviously not feasible.

3. $\frac{\sqrt{17}+1}{8} < \lambda < \frac{\sqrt{3}+1}{2}$: In this case, the solution of [85] is better than any solution using a single installment per load, but it may require a very large number of installments. This case is illustrated by Figure 2.4 with $\lambda = 1$.

   Now, let us compute the number of installments if $\lambda \neq 1$. We know that the i-th installment is equal to $\beta_i = \lambda^i \gamma_2^1(1)$, except the last one, which can be smaller than $\lambda^Q \gamma_2^1(1)$. So, instead of writing $\sum_{i=1}^{Q} 2\beta_i = \left(\sum_i^{Q-1} 2\lambda^i \gamma_2^1(1)\right) + 2\beta_Q = 1$, we write:

$$\sum_{i=1}^{Q} 2\lambda^i \gamma_2^1(1) \geq 1 \Leftrightarrow \frac{2\lambda^2\left(\lambda^Q - 1\right)}{(\lambda-1)(2\lambda+1)} \geq 1$$
$$\Leftrightarrow \frac{2\lambda^{Q+2}}{(\lambda-1)(2\lambda+1)} \geq \frac{2\lambda^2}{(\lambda-1)(2\lambda+1)} + 1.$$

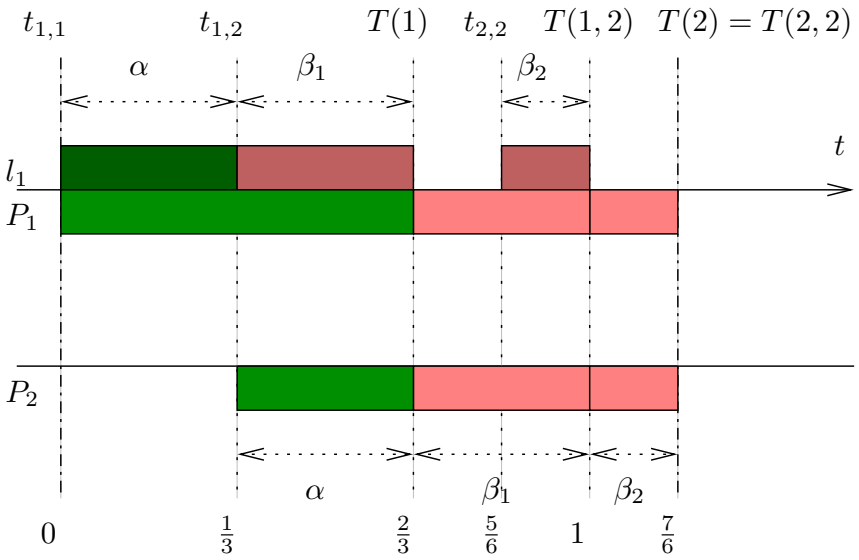


Figure 2.4: The example with $\lambda = 1$, $\alpha = \gamma_2^1(1)$ and $\beta_l = \gamma_2^l(2)$.

If λ is strictly smaller than 1, we obtain:

$$\begin{aligned}
& \tfrac{2\lambda^{Q+2}}{(\lambda-1)(2\lambda+1)} && \geq \frac{2\lambda^2}{(\lambda-1)(2\lambda+1)} + 1. \\
\Leftrightarrow \quad & 2\lambda^{Q+2} && \leq 4\lambda^2 - \lambda - 1 \\
\Leftrightarrow \quad & \ln(\lambda^Q) && \leq \ln\left(\frac{4\lambda^2-\lambda-1}{2\lambda^2}\right) \\
\Leftrightarrow \quad & Q\ln(\lambda) && \leq \ln\left(\frac{4\lambda^2-\lambda-1}{2\lambda^2}\right) \\
\Leftrightarrow \quad & Q && \geq \frac{\ln\left(\frac{4\lambda^2-\lambda-1}{2\lambda^2}\right)}{\ln(\lambda)}.
\end{aligned}$$

We thus obtain:

$$Q = \left\lceil \frac{\ln\left(\frac{4\lambda^2-\lambda-1}{2\lambda^2}\right)}{\ln(\lambda)} \right\rceil.$$

When λ is strictly greater than 1 we obtain the exact same result (then $\lambda - 1$ and $\ln(\lambda)$ are both positive).

To see that this choice is not optimal, consider the case $\lambda = \frac{3}{4}$. The algorithm of [85] achieves a makespan equal to $\left(1 - \gamma_2^1(1)\right)\lambda + \frac{\lambda}{2} = \frac{9}{10}$. The first load is sent in one installment and the second one is sent in 3 installments (according to the previous equation).

However, we can come up with a better schedule by splitting both loads into two installments, and distributing them as follows:

- during the first round, P_1 processes 0 unit of the first load,
- during the second round, P_1 processes $\frac{317}{653}$ unit of the first load,

- during the first round, P_2 processes $\frac{192}{653}$ unit of the first load,
- during the second round, P_2 processes $\frac{144}{653}$ unit of the first load,
- during the first round, P_1 processes 0 unit of the second load,
- during the second round, P_1 processes $\frac{464}{653}$ unit of the second load,
- during the first round, P_2 processes $\frac{108}{653}$ unit of the second load,
- during the second round, P_2 processes $\frac{81}{653}$ unit of the second load.

This scheme gives us a total makespan equal to $\frac{781}{653}\frac{3}{4} \approx 0.897$, which is (slightly) better than 0.9. This shows that among the schedules having a total number of four installments, the solution of [85] is suboptimal.

If $\lambda = 1$, we have:

$$\sum_{i=1}^{Q} 2\lambda^i \gamma_2^1(1) \geq 1,$$

simply leading us to $Q = 2$ and to a feasible solution.

Thus, this algorithm may lead to unfeasible solutions when communications are large, and to suboptimal solutions in case of small communication times.

2.3.4 Conclusion

Despite its simplicity (two identical processors and two identical loads), the analysis of this illustrative example clearly outlines the limitations of the approach of [85]: this approach does not always return a feasible solution and, when it does, this solution is not always optimal.

As we said before, the main drawback of the previous approach is to search for local optimums. The authors of [85], by forcing each load to finish at the same time on all processors, designed their solution as if the optimality principle, which is only true for a single load, was true for several loads. Moreover, they wanted to remove, on each processor, any potential computation idle time between the processing of two consecutive loads. However, these constraints are useless to obtain a valid schedule, but can artificially limit the solution space.

2.4 Optimal solution

In this section we show how to compute an optimal schedule, when dividing each load into any prescribed number of installments. We will discuss the computation of the right number of installments in Section 2.5.

When the number of installments is set to 1 for each load (i.e., $Q_k = 1$, for any k in $[1, m]$), the following approach solves the problem originally targeted by Min, Veeravalli, and Barlas.

To build our solution we use a linear programming approach. In fact, we only have to list all the (linear) constraints that must be fulfilled by a schedule, and write that we want to minimize the *makespan*.

All these constraints are captured by the linear program (2.2). This linear program simply encodes the following constraints (where a number in parentheses is the number of the corresponding constraint of linear program (2.2)):

- P_i cannot start a new communication to P_{i+1} before the end of the corresponding communication from P_{i-1} to P_i (2.2a),

- P_i cannot start to receive the next installment of the k-th load before having finished to send the current one to P_{i+1} (2.2b),
- P_i cannot start to receive the first installment of the next load before having finished to send the last installment of the current load to P_{i+1} (2.2c),
- any transfer has to begin at a nonnegative time (2.2d),
- the duration of any transfer is equal to the product of the time taken to transmit a unit load (2.2e) by the volume of data to transfer,
- processor P_i cannot start to compute the j-th installment of the k-th load before having finished to receive the corresponding data (2.2f),
- the duration of any computation is equal to the product of the time taken to compute a unit load (2.2g) by the volume of computations,
- processor P_i cannot start to compute the first installment of the next load before it has completed the computation of the last installment of the current load (2.2h),
- processor P_i cannot start to compute the next installment of a load before it has completed the computation of the current installment of that load (2.2i),
- processor P_i cannot start to compute the first installment of the first load before its availability date (2.2j),
- every portion of a load dedicated to a processor is necessarily nonnegative (2.2k),
- any load has to be completely processed (2.2l),
- the *makespan* is no smaller than the completion time of the last installment of the last load on any processor (2.2m).

$$
\left\{
\begin{array}{l}
\text{MINIMIZE } \textit{makespan} \text{ UNDER THE CONSTRAINTS} \\
\text{(2.2a)} \quad \forall i < n-1, k \leq m, j \leq Q_k, \quad Comm^{start}_{i+1,k,j} \geq Comm^{end}_{i,k,j} \\
\text{(2.2b)} \quad \forall i < n-1, k \leq m, j < Q_k, \quad Comm^{start}_{i,k,j+1} \geq Comm^{end}_{i+1,k,j} \\
\text{(2.2c)} \quad \forall i < n-1, k < m, \quad Comm^{start}_{i,k+1,1} \geq Comm^{end}_{i+1,k,Q_k} \\
\text{(2.2d)} \quad \forall i \leq n-1, k \leq m, j \leq Q_k, \quad Comm^{start}_{i,k,j} \geq 0 \\
\text{(2.2e)} \quad \forall i \leq n-1, k \leq m, j \leq Q_k, \quad Comm^{end}_{i,k,j} = Comm^{start}_{i,k,j} + \dfrac{V_{comm}(k)}{bw_i} \displaystyle\sum_{l=i+1}^{n} \gamma_l^j(k) \\
\text{(2.2f)} \quad \forall i \geq 2, k \leq m, j \leq Q_k, \quad Comp^{start}_{i,k,j} \geq Comm^{end}_{i,k,j} \\
\text{(2.2g)} \quad \forall i \leq n, k \leq m, j \leq Q_k, \quad Comp^{end}_{i,k,j} = Comp^{start}_{i,k,j} + \dfrac{\gamma_i^j(k) V_{comp}(k)}{s_i} \\
\text{(2.2h)} \quad \forall i \leq n, k < m, \quad Comp^{start}_{i,k+1,1} \geq Comp^{end}_{i,k,Q_k} \\
\text{(2.2i)} \quad \forall i \leq n, k \leq m, j < Q_k, \quad Comp^{start}_{i,k,j+1} \geq Comp^{end}_{i,k,j} \\
\text{(2.2j)} \quad \forall i \leq n, \quad Comp^{start}_{i,1,1} \geq \tau_i \\
\text{(2.2k)} \quad \forall i \leq n, k \leq m, j \leq Q_k, \quad \gamma_i^j(k) \geq 0 \\
\text{(2.2l)} \quad \forall k \leq m, \quad \displaystyle\sum_{i=1}^{n} \sum_{j=1}^{Q} \gamma_i^j(k) = 1 \\
\text{(2.2m)} \quad \forall i \leq n, \quad makespan \geq Comp^{end}_{i,m,Q}
\end{array}
\right. \tag{2.2}
$$

Lemma 2.1. *Consider, under a linear cost model for communications and computations, an*

instance of our problem with one or more load, at least one processor, and a given maximum number of installments for each load. If, as in [84, 85], loads have to be sent in the order of their submission, then the linear program (2.2) finds a valid and optimal schedule.

Proof. First, we can ensure that the provided schedule is valid:

- all starting time and installment sizes are nonnegative (2.2d, 2.2k),
- each computation only begins after the reception of the corresponding data (2.2f),
- at most one computation is processed at any time on any processor, and installments are processed following the submission order (2.2g, 2.2h, 2.2i, 2.2j),
- any load is completely processed (2.2l),
- all communications respect the strict one-port model and the submission order (2.2a, 2.2b, 2.2c, 2.2e).

The only non-essential constraint is the respect of the submission order by the computations (which is imposed by (2.2g), (2.2h), (2.2i)), since we could have inverted the computation of two installments on the same processor. This constraint allows the linear program to give a complete description of the schedule, with starting and ending time for any computation and any communication.

Moreover, this constraint does not change the minimum makespan: we know that an optimal algorithm to the problem described as $1|r_j|C_{max}$ (minimizing the makespan on one machine with release dates) in [30, p. 63] is the classical FCFS (First Come, First Served) algorithm. Thus imposing to process tasks according to their the submission order on a processor does not change the total computation time.

Since all other constraints are essential to have a valid schedule, we can assert than the schedule obtained by finding an optimal solution to the linear program is an optimal schedule. ■

Altogether, we have a linear program to be solved over the rationals, hence a solution in polynomial time [58]. In practice, standard packages like Cplex citecplex, Maple [33] or GLPK [49] will return the optimal solution for all reasonable problem sizes.

Note that the linear program gives the optimal solution for a prescribed number of installments for each load. We will discuss the problem of the number of installments in Section 2.5.

2.5 Possible extensions

There are several restrictions in the model of [85] that can be alleviated. First the model uses *uniform machines*, meaning that the speed of a processor does not depend on the task that it executes. It is easy to extend the linear program for unrelated machines, introducing s_i^k to denote the speed of P_i while processing a part of load k. Also, all processors and loads are assumed to be available from the beginning. In our linear program, we have introduced availability dates for processors. In the same way, we could have introduced release dates for loads. Furthermore, instead of minimizing the makespan, we could have targeted any other objective function which is an affine combination of the load completion times and of the problem characteristics, like the average completion time, the maximum or average (weighted) flow, etc.

The formulation of the problem does not allow any piece of the k'-th load to be processed before the k-th load is completely processed, if $k' > k$. We can easily extend our solution to

allow for m rounds of the m loads, each load being still divided into several installments. This would allow to interleave the processing of the different loads.

The divisible load model is linear, which causes major problems for multi-installment approaches. Indeed, once we have a way to find an optimal solution when the number of installments per load is given, the question is: what is the optimal number of installments? Under a linear model for communications and computations, the optimal number of installments is infinite, as the following theorem states:

Theorem 2.1. *Consider, under a linear cost model for communications and computations, an instance of our problem with one or more load and at least two processors, such that all processors are initially idle. Then, any schedule using a finite number of installments is suboptimal for makespan minimization.*

Proof. This theorem is proved by building, from any schedule using a finite number of installments, another schedule with a strictly smaller makespan.

We first remark that in any optimal solution to our problem all processors work and complete their share simultaneously. To prove this statement, we consider a schedule where one processor completes its share strictly before the makespan (this processor may not be doing any work at all). Then, under this schedule there exists two neighbor processors, P_i and P_{i+1}, such that one finishes at the makespan, denoted $\mathcal{M}$, and one strictly earlier. We have two cases to consider:

1. There exists a processor P_i which finishes strictly before the makespan $\mathcal{M}$ and such that the processor P_{i+1} completes its share exactly at time $\mathcal{M}$. P_{i+1} receives all the data it processes from P_i. We consider any installment j of any load L_k that is effectively processed by P_{i+1} (that is, P_{i+1} processes a non null portion of the j-th installment of load L_k and processes nothing hereafter). We modify the schedule as follows: P_i enlarges by an amount ε, and P_{i+1} decreases by an amount ε, the portion of the j-th installment of the load L_k it processes. Then, the completion time of P_i is increased, and that of P_{i+1} is decreased, by at least an amount proportional to ε as our cost model is linear. More precisely, the completion time of P_i is increased by an amount equal to $\varepsilon V_{comp}(k)/s_i$ and the completion time of P_{i+1} is decreased by an amount between $\varepsilon V_{comp}(k)/s_{i+1}$ and $\varepsilon(V_{comm}(k)/bw_i + V_{comp}(k)/s_{i+1})$.

   If ε is small enough, both processors complete their work strictly before $\mathcal{M}$. With our modification of the schedule, the size of a single communication was modified, and this size was decreased. Therefore, this modification did not enlarge the completion time of any processor except P_i. Therefore, the number of processors whose completion time is equal to $\mathcal{M}$ is decreased by at least one by our schedule modification.

2. No processor which completes it share strictly before time $\mathcal{M}$ is followed by a processor finishing at time $\mathcal{M}$. Therefore, there exists an index i such that the processors P_1 through P_i all complete their share exactly at $\mathcal{M}$, and the processors P_{i+1} through P_n complete their share strictly earlier. Then, let the last processing of processor P_i be that of installment j of load L_k. We have $Comp^{end}_{i+1,j,k}, \ldots, Comp^{end}_{n,j,k} < \mathcal{M}$.

   Then P_i decreases by a size ε, and P_{i+1} increases by a size ε, the portion of the j-th installment of load L_k that it processes.

   Then the completion time of P_i is decreased by an amount $\varepsilon V_{comp}(k)/s_i$, thus proportional to ε. The completion time of processor P_{i+1} is increased by at most $\varepsilon(V_{comp}(k)/s_{i+1} +$

$V_{comm}(k)/bw_i$), while the completion times of the processors P_{i+2} through P_n is at most increased by an amount $\varepsilon V_{comm}(k)/bw_i$, proportional to ε.

Let A be equal to:

$$\max\left(\max_{i+1\leq u\leq n}\left(\frac{\mathcal{M}-Comp^{end}_{u,j,k}}{V_{comm}(k)/bw_i}\right), \frac{\mathcal{M}-Comp^{end}_{i+1,j,k}}{V_{comm}(k)/bw_i+V_{comp}(k)/s_i}\right).$$

Therefore, if ε is small enough (i.e., $0<\varepsilon<A$), the processors P_i through P_n complete their work strictly before $\mathcal{M}$.

In both cases, after we modified the schedule, there is at least one more processor which completes its work strictly before time $\mathcal{M}$, and no processor is completing its share after that time. If no processor is any longer completing its share at time $\mathcal{M}$, we have obtained a schedule with a better makespan. Otherwise, we just iterate our process. As the number of processors is finite, we will eventually end up with a schedule whose makespan is strictly smaller than $\mathcal{M}$. Hence, in an optimal schedule all processors complete their work simultaneously (and thus all processors work).

We now prove the theorem itself by contradiction. Let $\mathcal{S}$ be any optimal schedule using a finite number of installments. As processors P_2 through P_n initially hold no data, they stay temporarily idle during the schedule execution, waiting to receive some data to be able to process them. Let us consider processor P_2. As the idleness of P_2 is only temporary (all processors are working in an optimal solution), this processor is only idle because it is lacking data to process and it is waiting for some. Therefore, the last moment at which P_2 stays temporarily idle under $\mathcal{S}$ is the moment it finished to receive some data, namely the j_0-th installment of load L_{k_0} sent to him by processor P_1.

As previously, Q_l is the number of installments of the load L_l under $\mathcal{S}$. Then from the schedule $\mathcal{S}$ we build a schedule $\mathcal{S}'$, identical to $\mathcal{S}$ except that we replace the j_0-th installment of load L_{k_0} by two new installments. The replacement of the j_0-th installment of load L_{k_0} only affects processors 1 and 2: for the others the first new installment brings no work to process and the second brings exactly the same amount of work than the j_0-th installment of load L_{k_0} in $\mathcal{S}$. Formally, using the same notations for $\mathcal{S}'$ than for $\mathcal{S}$, but with an added prime, $\mathcal{S}'$ is defined as follows:

- All loads except L_{k_0} have the exact same installments under $\mathcal{S}'$ than under $\mathcal{S}$: $\forall k \in [1,m]\setminus\{k_0\}$, $Q'_k = Q_k$ and $\forall i\in[1,n]$, $\forall j\in[1,Q_k]$, $\gamma'^{j}_i(k)=\gamma^j_i(k)$.
- The load L_{k_0} has $Q'_{k_0}=(1+Q_{k_0})$ installments under $\mathcal{S}'$, defined as follows:
  - The first (j_0-1) installments of L_{k_0} under $\mathcal{S}'$ are identical to the first $(j-1)$ installments of this load under $\mathcal{S}$: $\forall i\in[1,n]$, $\forall j\in[1,j_0-1]$, $\gamma'^{j}_i(k_0)=\gamma^j_i(k_0)$.
  - Installment j_0 of L_{k_0} is defined as follows:
    $\gamma'^{j_0}_1(k_0)=\gamma^{j_0}_1(k_0)$,
    $\gamma'^{j_0}_2(k_0)=\frac{1}{2}\gamma^{j_0}_2(k_0)$,
    $\forall i\in[3,n], \gamma'^{j_0}_i(k_0)=0$.
  - Installment j_0+1 of L_{k_0} is defined as follows:
    $\gamma'^{j_0+1}_2(k_0)=0$,
    $\gamma'^{j_0+1}_2(k_0)=\frac{1}{2}\gamma^{j_0}_2(k_0)$,
    $\forall i\in[3,n], \gamma'^{j_0+1}_i(k_0)=\gamma^{j_0}_i(k_0)$.

– The last $(Q_{k_0} - j_0)$ installments of L_{k_0} under $\mathcal{S}'$ are identical to the last $(Q_{k_0} - j_0)$ installments of this load under $\mathcal{S}$: $\forall i \in [1, n]$, $\forall j \in [j_0 + 1, Q'_{k_0}]$, $\gamma_i'^{j}(k_0) = \gamma_i^{j-1}(k_0)$.

Since the j_0-th installment of the k_0-th load is the first modified one, starting and ending times of each previous installment remain unchanged:

$$\begin{aligned}
\forall k < k_0 \text{ and } \forall j \in [1, Q_k] \text{ or } k = k_0 \text{ and } \forall j \in [1, j_0 - 1], \\
\forall i \in [1, n-1], Comm'^{\,start}_{i,k,j} = Comm^{start}_{i,k,j}, \\
\forall i \in [1, n-1], Comm'^{\,end}_{i,k,j} = Comm^{end}_{i,k,j}, \\
\forall i \in [1, n], Comp'^{\,start}_{i,k,j} = Comm^{start}_{i,k,j}, \\
\forall i \in [1, n], Comp'^{\,end}_{i,k,j} = Comp^{end}_{i,k,j}.
\end{aligned} \tag{2.3}$$

Now, let us focus on the j_0-th installment. We can easily derive the following properties for the first processor:

$$\begin{aligned}
Comp'^{\,start}_{1,k_0,j_0} &= Comp^{start}_{1,k_0,j_0}, \\
Comp'^{\,end}_{1,k_0,j_0} &= Comp^{end}_{1,k_0,j_0}, \\
Comp'^{\,start}_{1,k_0,j_0+1} &= Comp^{end}_{1,k_0,j_0}, \\
Comp'^{\,end}_{1,k_0,j_0+1} &= Comp^{end}_{1,k_0,j_0}.
\end{aligned}$$

We can write the following equations about the communication between P_1 and P_2 :

$$Comm'^{\,start}_{1,k_0,j_0} = Comm^{start}_{1,k_0,j_0},$$

$$Comm'^{\,end}_{1,k_0,j_0} = Comm^{start}_{1,k_0,j_0} + \frac{1}{2}\gamma_1^{j_0}(k_0) * V_{comm}(k_0)/bw_1, \tag{2.4}$$

$$Comm'^{\,start}_{1,k_0,j_0+1} = Comm'^{\,end}_{1,k_0,j_0},$$

$$Comm'^{\,end}_{1,k_0,j_0+1} = Comm^{end}_{1,k_0,j_0}. \tag{2.5}$$

There are only two constraints on the beginning of the computation on P_2 :

$$Comp'^{\,start}_{2,k_0,j_0} = \max\left\{Comm'^{\,end}_{1,k_0,j_0}, Comp'^{\,end}_{2,k_0,j_0-1}\right\}, \tag{2.6}$$

$$Comp^{start}_{2,k_0,j_0} = \max\left\{Comm^{end}_{1,k_0,j_0}, Comp^{end}_{2,k_0,j_0-1}\right\}. \tag{2.7}$$

Of course, Equations (2.6) and (2.7) are only true for $j_0 > 1$, we have to replace $Comp'^{\,end}_{2,k_0,j_0-1}$ (respectively $Comp^{end}_{2,k_0,j_0-1}$) by $Comp'^{\,end}_{2,k_0-1,Q_{k_0-1}}$ (respectively $Comp^{end}_{2,k_0-1,Q_{k_0-1}}$) if we have $k_0 > 1$ and $j_0 = 1$, and by 0 in both cases if $k_0 = 1$ and $j_0 = 1$, we recall that all processors are initially idle[1].

[1] This constraint is a bit too strong. The theorem is still true when only one processor (different from P_1) is initially idle. If all processors have strictly positive release times, they can finish their first communication and immediately start to compute the first installment of the first load, without any idle time between their release date and their first computation, and our theorem is false.

By definition of j_0 and k_0, P_2 is idle right before the beginning of the computation of the j_0-th installment of the k_0-th load, therefore:

$$Comp^{end}_{2,k_0,j_0-1} < Comm^{end}_{1,k_0,j_0}. \tag{2.8}$$

Using Equation (2.7), we thus have:

$$Comp^{start}_{2,k_0,j_0} = Comm^{end}_{1,k_0,j_0}. \tag{2.9}$$

Moreover, since we have $\gamma_2^{j_0}(k_0) > 0$, the communication of the j_0-th installment between P_1 and P_2 in $\mathcal{S}'$ ends strictly earlier than the communication of the j_0-th installment between these processors in $\mathcal{S}$:

$$Comm'^{\,end}_{1,k_0,j_0} < Comm^{end}_{1,k_0,j_0} = Comp^{start}_{2,k_0,j_0}. \tag{2.10}$$

We can apply Equation (2.3) for the (j_0-1)-th installment of the k_0 load, and use Equations (2.8) and (2.9):

$$Comp'^{\,end}_{2,k_0,j_0-1} = Comp^{end}_{2,k_0,j_0-1} < Comp^{start}_{2,k_0,j_0} \tag{2.11}$$

In our new schedule $\mathcal{S}'$, by using Equations (2.6), (2.10), and (2.11), we can say that the computation on the new j_0-th installment begins strictly earlier on P_2 :

$$Comp'^{\,start}_{2,k_0,j_0} < Comp^{start}_{2,k_0,j_0}, \tag{2.12}$$

$$Comp'^{\,start}_{2,k_0,j_0+1} = \max\left\{ Comp'^{\,end}_{2,k_0,j_0}, Comm'^{\,end}_{1,k_0,j_0+1} \right\}. \tag{2.13}$$

By definition of $Comp'^{\,end}_{2,k_0,j_0}$, and $Comp'^{\,end}_{2,k_0,j_0+1}$, we have the two following Equations (2.14) and (2.15):

$$Comp'^{\,end}_{2,k_0,j_0} = Comp'^{\,start}_{2,k_0,j_0} + \frac{Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,j_0,k_0}}{2}, \tag{2.14}$$

$$Comp'^{\,end}_{2,k_0,j_0+1} = Comp'^{\,start}_{2,k_0,j_0+1} + \frac{Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,j_0,k_0}}{2}. \tag{2.15}$$

If we use (2.15), (2.14), (2.13) and (2.5):

$$Comp'^{\,end}_{2,k_0,j_0+1} = \max \begin{cases} \frac{Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,j_0,k_0}}{2} + Comm^{end}_{1,k_0,j_0}, \\ Comp'^{\,start}_{2,k_0,j_0} + Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,k_0,j_0} \end{cases}. \tag{2.16}$$

Since we have Equation (2.8) and $0 < \gamma_2^{k_0}(j_0)$, we have $Comp^{end}_{2,k_0,j_0} > Comp^{start}_{2,k_0,j_0}$ and then

$$\begin{aligned} \frac{Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,j_0,k_0}}{2} + Comm^{end}_{1,k_0,j_0} < \\ Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,j_0,k_0} + Comm^{end}_{1,k_0,j_0} \\ = Comp^{end}_{2,k_0,j_0}. \end{aligned} \tag{2.17}$$

By using Equation (2.12), we can ensure:

$$Comp'^{\,start}_{2,k_0,j_0} + Comp^{end}_{2,k_0,j_0} - Comp^{start}_{2,k_0,j_0} < Comp^{end}_{2,k_0,j_0}. \tag{2.18}$$

By combining (2.17), (2.18) and (2.16), we have:

$$Comp'^{\,end}_{2,k_0,j_0+1} < Comp^{end}_{2,k_0,j_0}. \tag{2.19}$$

Therefore, under schedule $\mathcal{S}'$ processor P_2 completes strictly earlier than under $\mathcal{S}$ the computation of what was the j_0-th installment of load L_{k_0} under $\mathcal{S}$. If P_2 is no more idle after the time $Comp'^{\,end}_{2,k_0,j_0}$, then it completes its overall work strictly earlier under $\mathcal{S}'$ than under $\mathcal{S}$. P_1 completes its work at the same time. Then, using the fact that in an optimal solution all processors finish simultaneously, we conclude that $\mathcal{S}'$ is not optimal. As we have already remarked that its makespan is no greater than the makespan of $\mathcal{S}$, we end up with the contradiction that $\mathcal{S}$ is not optimal. Therefore, P_2 must be idled at some time after the time $Comp'^{\,end}_{2,k_0,j_0}$. Then we apply to $\mathcal{S}'$ the transformation we applied to $\mathcal{S}$ as many times as needed to obtain a contradiction. This process is bounded as the number of communications that processor P_2 receives after the time it is idle for the last time is strictly decreasing when we transform the schedule $\mathcal{S}$ into the schedule $\mathcal{S}'$. ■

An infinite number of installments obviously does not define a feasible solution. Moreover, in practice, when the number of installments becomes too large, the model is inaccurate, as acknowledged in [28, p. 224 and 276]. Any communication incurs a startup cost l, which we express in bytes. Consider the k-th load, whose communication volume is $V_{comm}(k)$: it is split into Q_k installments, and each installment requires $n-1$ communications. The ratio between the actual and estimated communication costs is roughly equal to $\rho = \frac{(n-1)Q_k l + V_{comm}(k)}{V_{comm}(k)} > 1$. Since l, n, and $V_{comm}(k)$ are known values, we can choose Q_k such that ρ is kept relatively small, and so such that the model remains valid for the target application. Another, and more accurate solution, would be to introduce latencies in the model, as in Section 1.3.1. This latter article shows how to design asymptotically optimal multi-installment strategies for star networks. A similar approach should be used for linear networks.

Note that Theorem 2.1 also holds true in case of star-shaped networks. We show it in [A1] by adapting the proof of Theorem 2.1.

Thus, any optimal solution uses many installments. However, the benefit of distributing the entire load using several installments can be bounded, as stated by the following theorem.

Theorem 2.2. *Consider, under a linear cost model for communications and computations, an instance of our problem with one or more load, such that all processors are initially idle. Then, the minimum makespan using a single installment for each load is less than* $(\lfloor n/2 \rfloor + 1)$ *times the optimal makespan obtained with several installments.*

Proof. Consider several loads distributed to a chain of n processors, using a schedule with Q installments. Let $makespan_Q$ be the makespan of this schedule, and let α_i^k be the fraction of the k-th load allocated to each processor: $\forall i \leq n, \alpha_i^k = \sum_{j=1}^{Q} \gamma_i^j(k)$.

Consider a new schedule using a single installment, such that any processor computes the same fraction α_i^k of each load. Let $makespan_1$ be the makespan of this solution.

By definition of $makespan_Q$, any processor P_i requires less than $makespan_Q$ time units to complete all its communications (receptions *and* transmissions) and less than $makespan_Q$ to complete its computations. Thus, P_2 has finished to receive its own data and to send the remaining share to P_3 before time $makespan_Q$. Similarly, P_4 can finish its incoming and outgoing communications before time $2 \times makespan_Q$, and so on: P_{2i} is able to finish its communications before time $i \times makespan_Q$ and P_{2i+1} has received its data at this time. More generally, P_j has

received its data before time $\lfloor j/2 \rfloor makespan_Q$ and can finish its computation at most at time $(\lfloor j/2 \rfloor + 1) makespan_Q$.

If T_i denotes the completion time of P_i, we have by definition of $makespan_1$:

$$makespan_1 = \max_{1 \leq i \leq n} T_i \leq \max_{1 \leq i \leq n} \left\{ \left(\left\lfloor \frac{i}{2} \right\rfloor + 1 \right) makespan_Q \right\} = \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) makespan_Q.$$

This concludes our proof. Note that our new schedule is suboptimal, since all processors do not finish at the same time.

To check whether this bound is tight, we consider a chain of n processors, and a single load of size 1: $V_{comm}(1) = V_{comp}(1) = 1$. The platform is defined by:

$$\begin{aligned} \forall i, 1 \leq i < n, s_i &= 0, \\ s_n &= 1/2, \\ \forall i, 1 \leq i < n, bw_i &= 1. \end{aligned}$$

Only the last processor can perform some work, and communication links are homogeneous.

There is only one valid single-installment schedule, whose makespan is easily computed: the entire load has to be sent in time $(n-1)$ to processor P_n, which finishes its work at time $makespan_1 = (n-1) + 2 = n + 1$.

Now, let us compute the makespan $makespan_Q$ of a schedule using Q identical installments, of size $1/Q$. Any communication from P_1 to P_n passes through $n-1$ links. Indeed, the first round is received by P_n at time $(n-1)/Q$, and is completely processed at time $(n-1+2)/Q = (n+1)/Q$. Due to the strict one-port model, the second round is sent by P_1 at time $2/Q$ and is received by P_n at time $(n+1)/Q$. Finally, the Q-th round (the last one) is received by P_n at time $(n-3+2Q)/Q$. P_m achieves its computation at time $makespan_Q = (n-1+2Q)/Q$.

Indeed, we have the following relation:

$$makespan_1 = makespan_Q \left(\frac{(n+1)\,Q}{n-1+2Q} \right).$$

Considering large numbers of installments leads to:

$$\lim_{Q \to \infty} \frac{makespan_1}{makespan_Q} = \frac{n+1}{2}.$$

Thus, the bound is tight with an infinite number of installments for odd numbers of processors. ■

2.6 Experiments

Using simulations, we now assess the relative performance of our linear programming approach, of the solutions of [84, 85], and of simpler heuristics. We first describe the experimental protocol and then analyze the results.

Experimental protocol. We use Simgrid [63] to simulate linear processor networks. Schedules are pre-computed by a script, and their validity and theoretical makespan are checked before running them in the simulator.

We study the following algorithms and heuristics:

- The naive heuristic SIMPLE distributes each load in a single installment and proportionally to the processor speeds.

- The strategy for a single load, SINGLELOAD, presented by Min and Veeravalli in [84]. For each load, we set the time origin to the availability date of the first communication link (in order to try to prevent communication contentions).

- The MULTIINST k strategy. The main strategy proposed by Min, Veeravalli and Barlas is to split each loads into several installments, in order to overlap communications by computations, and we called it MULTIINST. However, they do not fix any limit on the total number of installments, and MULTIINST k is a slightly modified version of MULTIINST which ensures that a load is not distributed in more than k installments, the $k-$-th installment of a load distributing all the remaining work of that load.

- The HEURISTIC B presented by Min, Veeravalli, and Barlas in [85].

- LP k : the solution of our linear program where each load is distributed in k installments.

We measure the relative performance of each heuristic on each instance: we divide the makespan obtained by a given heuristic on a given instance by the smallest makespan obtained, on that instance, among all heuristics. Considering the relative performance enables us to produce meaningful statistics among instances with very different makespans.

Instances. We emulate a heterogeneous linear network with $n = 10$ processors. We consider two distribution types for processing powers: *homogeneous* where each processor P_i has a processing power $s_i = 100$ MFLOPS, and *heterogeneous* where processing powers are uniformly picked between 10 and 100 MFLOPS. Communication link l_i has a bandwidth bw_i uniformly chosen between 10 Mb/s and 100 Mb/s, and a latency between 0.1 and 1 ms (links with high bandwidths having small latencies). For homogeneous and heterogeneous platforms, loads have their computation volumes either all uniformly distributed between 6 GFLOPS and 4 TFLOPS, or all uniformly distributed between 6 and 60 GFLOPS. For each combination of processing power distribution and task size, we fix the communication to computation volume of all tasks to either $0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50$, or 100 (bytes per FLOPS). Each instance contains 50 loads. Finally, we randomly built 100 instances per combination of the different parameters, hence a total of 3,600 instances simulated and reported in Table 2.2. The code and the experimental results can be downloaded from: `http://graal.ens-lyon.fr/~mgallet/downloads/DivisibleLoadsLinearNetwork.tar.gz`.

We only present the results of the simulation with Simgrid, without giving the pre-computed makespans (computed during the validity check of each schedule). Schedules were computed without latency according to the model. However, the communication model used for the simulation is realistic and thus includes latencies (see Section 2.5). Theses latencies are small, less than one millisecond as in many modern clusters. This is sufficient to have a small difference between predicted makespans and experimental ones, less than 1%, but since both values were very close, only experimental values are given.

We fixed an upper-bound to the number of installments per load used by the different heuristics: MULTIINST to either 100 or 300, SINGLELOAD to 100, and LP k to either 1, 2, 3, or 6.

Heuristic	Average	Std dev.	Max	Best result	Optimal solutions found
SIMPLE	1150.42	$1.6\ 10^{3}$	8385.94	3.66	0.00 %
SINGLELOAD 100	1462.65	$2.0\ 10^{3}$	10714.41	6.03	0.00 %
MULTIINST 100	1.13962	$1.8\ 10^{-1}$	1.98712	1.	7.64 %
MULTIINST 300	1.13963	$1.8\ 10^{-1}$	1.98712	1.	6.99 %
HEURISTIC B	1.13268	$1.7\ 10^{-1}$	2.01865	1.	4.72 %
LP 1	1.00047	$8.5\ 10^{-4}$	1.00498	1.	89.97 %
LP 2	1.00005	$9.6\ 10^{-5}$	1.00196	1.	97.32 %
LP 3	1.00002	$4.7\ 10^{-5}$	1.00098	1.	97.35 %
LP 6	1.00000	0	1.00001	1.	99.82 %

Table 2.2: Summary of results.

Discussions of the results. As we can see in Table 2.2, experimental values show that the linear program give almost always the best experimental makespan. There is a difference between pre-computed and experimental values, since LP 6 always give the best theoretical makespan but can be 0.01‰ away from the apparent best solution in the experimental results.

LP 1, LP 2, LP 3, and LP 6 achieve equivalent performance, always less than 5‰ away from the best result, and even LP 1 gives the best makespan in almost 90% of instances. This may seem counter-intuitive but can readily be explained: multi-installment strategies mainly reduce the idle time incurred on each processor before it starts processing the first task, and the room for improvement is thus quite small in our (and [85]) batches of 50 tasks. The strict one-port communication model forbids the overlapping of some communications due to different installments, and further limits the room for performance enhancement. Except in some peculiar cases, distributing the loads in multi-installments do not induce significant gains. In very special cases, LP 6 does not achieve the best performance during the simulations, but this fact can be explained by the latencies existing in simulations, and not taken into account in the linear program (2.2).

The bad performance of SIMPLE, which can have makespans 8000 greater than the optimal, justify the use of sophisticated scheduling strategies. The slight difference performance between MULTIINST 100 and MULTIINST 300 shows that MULTIINST sometimes uses a very large amount of installments for an insignificant negative gain (certainly due to latencies). When communication links are slow and when communications dominate computations, MULTIINST and HEURISTIC B can have makespans 98% higher than the optimal.

2.7 Conclusion

We have shown that a linear programming approach allows to solve all instances of the scheduling problem addressed in [84, 85]. In contrast, the original approach was providing a solution only for particular problem instances. Moreover, the linear programming approach returns an optimal solution for any given number of installments, while the original approach was empirically limited to very special strategies, and was often sub-optimal.

Intuitively, the solution of [85] is worse than the schedule of Section 2.3.1 because it aims at locally optimizing the makespan for the first load, and then optimizing the makespan for the second one, and so on, instead of directly searching for a global optimum. We did not find elegant closed-form expressions to characterize optimal solutions but, through the power of

linear programming, we have been able to find an optimal schedule for any instance.

Part II

Steady-state scheduling

Chapter 3

General presentation of steady-state scheduling

3.1 Introduction

In the first part, we studied the Divisible Load Theory, which is a common relaxation to a specific class of scheduling problems. In this second part, we focus on another relaxation scheme: we now target directed acyclic task graphs, of which a large number of instances has to be processed on a possibly heterogeneous computing platform. As said in Chapter , minimizing the total computation time (the makespan) is a really hard problem, even in very constrained cases. Moreover, this criteria can be unadapted to the actual situation.

In this part, we consider a large number of independent copies, or instances, of the same task graph, that we plan to process on a modern computing platform, such as a heterogeneous grid. This great number of copies allows to minimize the effect of small variations in resource capacities or in task characteristics. Today, such a situation happens in many contexts, for example when a continuous flow of information has to be processed as fast as possible. Due to the higher bandwidth of ADSL lines or even optic-fiber connexions, which are now widespread, multimedia applications like the real-time video broadcast of public events are more and more common, and require a lot of real-time processing from the digital video camera to the end user. More scientific applications also produce lots of data, which have to be processed on-the-fly before being stored; one could think of the 15 petabytes annually produced by the Large Hadron Collider (LHC) [42].

Obviously, we could use classical makespan minimization algorithms for this problem, but minimizing the total processing time of a continuous flow of data is nearly non-sense. Moreover, applying such general algorithms to such a problem does not permit to efficiently use its regular structure. In fact, we should take care of its regularity in the earliest stages of the algorithm design, to be sure to properly use it. In [16], Beaumont et al. have shown that the initialization and the end of the schedule have little importance as soon as the number of instances is large enough. This leads to only optimize the heart of the schedule, the steady-state phase, keeping out the start of the execution and its termination. Optimizing the steady-state phase means maximizing the number of instances of our task graph processed by the whole platform within T time units, i.e., the throughput, which will be denoted by ρ in the following pages. In several cases, maximizing the throughput ρ remains a tractable problem, while the makespan minimization turns out to be NP-hard in most practical situations [73, 9].

To be representative of actual computing grids, the platform model may be rather sophisti-

cated: such grids are often distributed over several physical locations, and are owned by different organizations and were bought at different times. Thus, the computing power of the nodes are strongly dependent of the processor architecture and their age. The same problem stands for communication times, and is even more complex since the same communication should not take the same time if both nodes are in the same cluster or on both sides of the Atlantic Ocean.

As said before, the regularity of our problem brought by the multiple copies of the same task graph is an important feature that we will take advantage of. The main idea behind the steady-state scheduling is to consider that after a reasonably short period of initialization, the throughput of each resource (either computing, or communicating resource) become stable, allowing us to deal with average numbers of tasks processed by resources during a single time unit, instead of working with integer numbers.

In Section 3.2, we expose the formulation of the problem, introducing notations common to all chapters of this part. Our solutions mainly use periodic steady-state schedules, which are presented in Section 3.3, while main differences between dynamic and static schedules are given in Section 3.4. Finally, Section 3.5 describes the different chapters of this part.

3.2 Problem formulation

In this part, we detail the modeling of the platform and the general form of the target applications.

3.2.1 Platform model

According to our need of modeling a great variety of heterogeneous platforms, we represent any platform by a graph $G_P = (V_P, E_P)$, where the set $V_P = \{P_1, \ldots, P_n\}$ of vertices corresponds to processors and the set E_P of edges to communication links. For the sake of simplicity, the word "resource" means either a processor (computation resource) or a link (communication resource), and other network resources like routers or switches are treated as processors without computational power.

A communication link $P_i \rightarrow P_j$ is characterized by its bandwidth $bw_{i,j}$, measured in B/s (byte per second). We restrain ourself to a linear communication model: the time to send a message is proportional to its size, and a message of size S needs a time $S/bw_{i,j}$ to be transfered from P_i to P_j. Since the exact communication and computation model is specific to each chapter, we do not present it here in depth. An short example of a 6-processor platform is shown in Figure 3.1. As we can see, the platform graph is not a clique and all processors are not directly connected, and any communication from P_1 to P_6 should pass through P_2 and P_5.

3.2.2 Application model

Steady-state techniques apply to a wide range of applications. However, we only focus our attention to applications that can be modeled by Directed Acyclic Graph (DAGs). We denote by $G_A = (V_A, E_A)$ the application graph, where $V_A = \{T_1, \ldots, T_m\}$ denotes the set of computing tasks, and E_A is the set of dependencies between these tasks, which are usually materialized by files: a task produces a file which is necessary for the processing of some other task. $F_{k,l} = (T_k \rightarrow T_l) \in E_A$ is the file produced by T_k and consumed by T_l. The dependency file $F_{k,l}$ has size $data_{k,l}$, so its transfer through link $P_i \rightarrow P_j$ takes a time $\frac{data_{k,l}}{bw_{i,j}}$. The computation time of each task follows an unrelated model, that is, a processor could be slower to process some

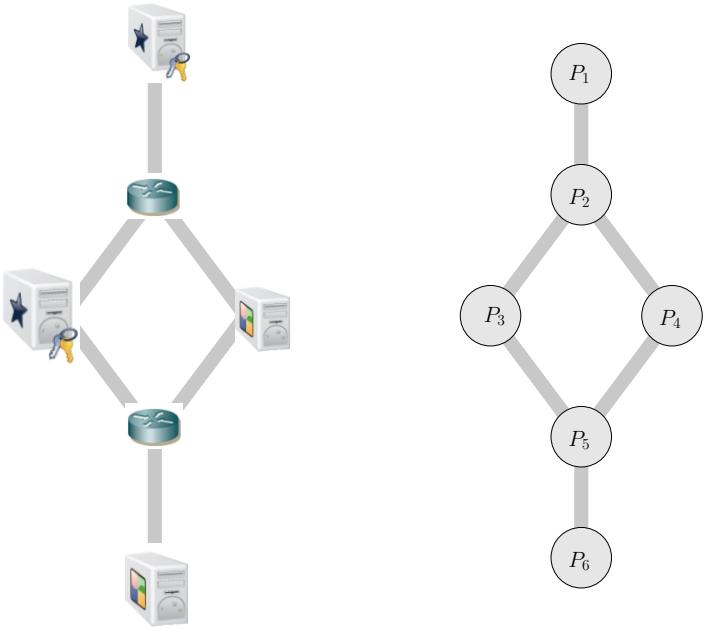


Figure 3.1: Example of platform graph, made of six processors, two of them being routers.

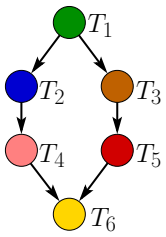


Figure 3.2: Example of application graph, made of 6 tasks.

tasks, and faster to compute some other tasks. Thus, we denote by $w_{i,k}$ the time needed by processor P_i to entirely process task T_k. Using these notations, we can model the benefits which can be drawn on specific hardware architectures by specially optimized tasks: certain types of tasks are especially well-suited to the vectorial units of processors like the Power5 of IBM. For example, a Cholesky factorization can be 5.5 times faster when using a GeForce 8800GTX graphic card than when using only the CPU, while a LU factorization is only 3 times faster in the same conditions [80]. Unrelated performance may also come from memory requirements. Indeed, a given task requiring a lot of memory will be completed faster when processed by a slower processor but with a larger amount of memory as soon as the hard drive is intensively used by the fast processor while the memory of the slow processor is large enough. Grids are often composed of several clusters bought over several years, thus with very different memory capacities, even if processors are rather similar.

N is the (large) number of instances of the graph G_A to be scheduled on the platform G_P. The notation $G_A^u = (V_A^u, E_A^u)$ refers to the u-th instance of G_A. Similarly, T_k^u is the u-th instance of T_k, and $F_{k,l}^u$ is the u-th instance of $F_{k,l}$. Figure 3.2 presents a simple example of an application DAG.

3.2.3 Definition of the allocations

Scheduling an application on a parallel platform requires at least to define *where* each task will be processed, i.e., which processor will execute it, and *when* the tasks will be computed. In this section, we only define an allocation of tasks to processors.

Definition 3.1 (allocation)**.** *An allocation $\sigma\left(G_A^u\right)$ of an instance u of the application graph G_A to the platform graph G_P is a function σ associating:*

- *to the instance u of each task T_k, a processor $P_i = \sigma\left(T_k^u\right)$,*
- *to the instance u of each file $F_{k,l}$, a set of communication links $\sigma\left(F_{k,l}^u\right)$, to transmit the file from $P_i = \sigma\left(T_k^u\right)$ to $P_j = \sigma\left(T_l^u\right)$.*

Since this definition remains general, we do not specify the routing policy used to transmit the communication from the source to the destination. Moreover, this definition only applies to a single instance of the task graph. Thus, two distinct instances of G_A may have different allocations. If a schedule uses the same allocation for all instances, then we simplify notations and $\sigma\left(G_A^u\right)$ denotes a function associating a processor $\sigma\left(T_k\right)$ to each task T_k of the application graph, and a set of links $\sigma\left(F_{k,l}\right)$ to each a file $F_{k,l}$.

3.3 Periodic steady-state scheduling

First, we formally define what we call the "throughput", that is the average number of instances that can be processed per time-unit in steady state.

Definition 3.2 (throughput)**.** *Assume that the number of instances to be processed is infinite, and let $N(t)$ be the number of instances totally processed by a schedule at time t. The throughput ρ of this schedule is given by $\rho = \lim_{t\to\infty} \frac{N(t)}{t}$.*

As explained in the previous section, we only focus our attention on the steady-state phase of the schedule, and, to preserve the simplicity of the studied problem, we look for periodic schedules. Given a platform G_P and a target application G_A, there is a finite number, even if it may be very large, of possible allocations: if we consider a homogeneous, fully-connected platform, there are at least n^m allocations of tasks to processors, without taking care of communications. The complete schedule of all the N instances of G_A could be very complicated, made by a large subset of interleaved allocations. This situation leads us to look for simpler, periodic schedules. They are made of a given pattern of allocations, which is repeated every $\mathcal{T}$ time units, ensuring a regular structure to the schedule:

Definition 3.3 (periodic schedule)**.** *A periodic schedule of period (or length) $\mathcal{T}$ is a set of n_T allocations $\{\sigma_1, \ldots, \sigma_{n_T}\}$, such that any resource can process its workload (either communications or computations) in at most $\mathcal{T}$ time units.*

Due to this periodicity, the average time spent on a given task by each resource is easily computed. In [16], Beaumont et al. exposed a general algorithm to build from these average values a complete schedule ensuring the same throughput $n_T/\mathcal{T}$, without knowledge on the original allocations and taking care of dependencies. Thus, only the average behavior of each resource needs to be specified by a schedule algorithm to produce a valid steady-state schedule. However, in the general case, the number of used allocations can be exponential in the size of the instance. Two examples of allocations of the same application graph G_A on the same platform are given in Figure 3.3.

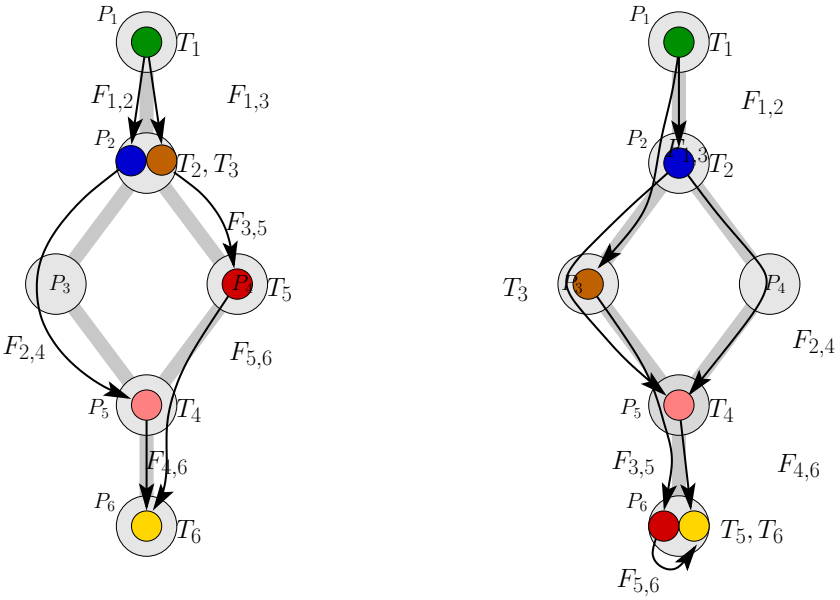


Figure 3.3: Two possible allocations of the same task graph.

3.4 Dynamic vs. static scheduling

Many scheduling strategies use a *dynamic* approach: task graphs, or even tasks, are processed one after the other. This is usually done by assigning priorities to waiting tasks, and then by allocating resources to the task with highest priority, as long as there are free resources. This simple strategy is the best possible in some cases: (i) when we have no knowledge on the future workload (i.e., the tasks that will be submitted in the near future, or released by the processing of current tasks), or (ii) under a very unstable environment, where machines join and leave the system with a high turnout rate.

Contrarily to the typical use of dynamic schedulers, we have more knowledge on the system when scheduling several instances of a given application. First, we can take advantage of the regularity of the pending jobs: the input is made of a large collection of data sets, on which the same treatments are applied, resulting in the same task graphs. Second, the computing platform is considered to be stable enough so that we can use performance measurement tools like NWS [83] in order to get some information on machine speeds and link bandwidths. Taking advantage of this knowledge, we aim at using *static* scheduling techniques, that is to anticipate the mapping and the scheduling of the whole workload at its submission date. Since periodic schedules are computed before the actual execution, one can check whether its platform is able to ensure a given throughput, and whether the memory requirements are fulfilled. Thus, both memory and throughput constraints can be guaranteed by a periodic schedule, offering a strong advantage against dynamic ones.

3.5 Content of this part

The term "steady-state scheduling" covers a wide range of different problems, several ones of them being studied in the next chapters. The fundamental idea of using steady-state techniques in scheduling comes from a article of Bertsimas and Gamarnik [25], mainly focused on routing

communication packets in computer networks. This approach has been successfully applied to problems as diverse as pipelining broadcasts [18], scheduling independent tasks [12] or divisible loads [54] on heterogeneous platforms. A steady-state approach for scheduling collections of identical task graphs was proposed in [19]. The solution given in [19], although asymptotically optimal under reasonable assumptions, may not be practical by reason of the very large number of involved allocations. Periodic schedules may be composed of several allocations, as exposed in Section 3.3. However, dealing with many allocations can be a drawback, mainly due to the difficulty of precisely controlling the data flows. Thus, in Chapter 4, we study the problem of finding efficient schedules made of a single allocation. Due to the intrinsic complexity of this problem, we also provide several heuristics.

In Chapter 5, the problem under study is somewhat different: instead of working with many instances of the same task graph, we want to schedule several collections of independent, heterogeneous tasks. The characteristics of these tasks are given by probability laws, and are grouped in collections following these laws. Thus, we can compare the results of smart approximations of the optimal schedule to simple schedule policies like Round-Robin and on-demand.

In Chapter 6, application graphs are restrained to simple pipelines, mapped on a heterogeneous platform following several allocations. If the Round-Robin policy ensures a simple control, determining the period $\mathcal{T}$ of the system is a complex problem, even if a complete allocation is given. Modeling this problem using timed Petri nets permits to establish this period for several communication models.

The next chapter, Chapter 7, is devoted to a small-scale, practical application of these steady-state scheduling techniques to the Cell processor. Built around a classical PowerPC core and height simpler cores called SPEs, the Cell processor is an innovative heterogeneous CPU designed by IBM as a new way to obtain high-performance processors. However, efficiently mapping an application graph on these different cores is really hard. To alleviate this constraint and ease the development of complex programs, we modeled this problem as a classical steady-state problem.

Chapter 4

Mono-allocation schedules of task graphs on heterogeneous platforms

4.1 Introduction

In this chapter, we investigate the problem of mapping an application onto the computing platform, being interested both in optimizing the performance of the mapping (that is, process the data as fast as possible), and in keeping the deployment simple, so that we do not have to deploy complex control softwares on a large number of machines.

We only consider *Grid jobs* made of the same workflow applied to a large collection of different input data sets, or, in other words, these *Grid jobs* are constituted of a large number of instances of the same task graph. Therefore, the context presented in Chapter 3 is perfectly matched. We already saw that there are many ways to map a single instance to the platform, and that a single periodic schedule can be decomposed in many allocations. The control system to ensure that all dependency files are sent to right processors is complex to deploy. Thus, we concentrate on how to compute periodic schedules from a single allocation, allowing lighter flow-control systems.

In the context of scheduling series of task graphs, we can take advantage of two sources of parallelism to increase performance. First, parallelism comes from the *data*, as we have to process a large number of instances. Second, each instance consists in a task graph which may well include some parallelism: some tasks can be processed simultaneously, or the processing of consecutive tasks of different instances can be pipelined, using some *control* parallelism. In such a context, several scheduling strategies may be used.

We may only make use of data parallelism. Then, the whole workflow corresponding to the processing of a single input data set is executed on a single resource, as if it was a large sequential task. Different workflow instances are simultaneously processed on different processors. This is potentially the solution with the best degree of parallelism, because it may well use all available resources. This imposes that all tasks of a given instance are performed on each processor, therefore that all services must be available on each participating machine. However, it is likely that some services have heterogeneous performance: many legacy codes are specialized for specific architectures and would perform very poorly if run on other machines. Some services are even likely to be unavailable on some machines. In the extreme, most specified case, it may happen that no machine can run all services; in that case the pure data-parallelism approach is infeasible. Moreover, switching, on the same machine, from one service to another may well induce some latency to deploy each service, thus leading to a large overhead. Finally, a single

input data set may well produce a large volume of data to process or require a large amount of memory. Processing the whole workflow on a single machine may lead to a large latency for this instance, and may even not be possible if the available storage capacity or memory of the machine cannot cope with the workflow requirements. For these reasons, application workflows are usually not handled using a pure data-parallelism approach.

Another approach consists in simultaneously taking advantage of both data and control parallelism. In [15, 17], it is proved that in a large number of cases, when the application graph is not too deep, we can compute an optimal schedule, that is a schedule which maximizes the system throughput. This approach, however, asks for a lot of control as similar files produced by different data sets must follow different similar files produced by different data sets must follow different paths in the interconnection network. In this chapter, we focus on a simpler framework: we aim at finding a single mapping of the application workflow onto the platform. This means that all instances of a given task must be processed on the same machine. Thus, the corresponding service has to be deployed on a single machine, and all instances are processed the same way. Thus, the control of the Grid job is much simpler, and the number of needed resources is kept low.

4.2 Notations, hypotheses, and complexity

In Section 3.2, we presented most of used notations, except the communication model. Here we complete this description, and we present the complexity of our problem.

4.2.1 Platform and application model

As said before, we denote by $G_P = (V_P, E_P)$ the undirected graph representing the platform. The edges of E_P represent the communication links between these processors. The maximum bandwidth of the communication link $P_q \to P_r$ is denoted by $bw_{q,r}$. Moreover, we suppose that processor P_q has a maximum incoming bandwidth bw_q^{in} and a maximum outgoing bandwidth bw_q^{out}. Figure 4.2(a) gives an example of such a platform graph. A path from processor P_q to processor P_r, denoted $P_q \rightsquigarrow P_r$, is a set of adjacent communication links going from P_q to P_r.

In this chapter, we use a bidirectional multiport model for communications: a processor can perform several communications simultaneously. In other words, a processor can simultaneously send data to multiple targets and receive data from multiple sources, as long as the bandwidth limitation is exceeded neither on links, nor on incoming or outgoing ports. The computation model is unrelated: a processor can be fast to process a given type of task, while being slow to execute another type of task. We denote by $w_{i,k}$ the time required by processor P_i to process a single instance of task T_k.

4.2.2 Allocations

Our definition of allocation given in Subsection 3.2.3 does not comprise the communication scheme, since a file $F_{i,j}$ may be transfered differently from $\sigma(T_i)$ to $\sigma(T_j)$ depending on the routing policy enforced on the platform. We distinguish three possible policies:

Single path, fixed routing. The path for any transfer from P_q to P_r is fixed a priori. We do not have any freedom on the routing. This scenario corresponds to the classical case where we have no freedom on the routing between machines: we cannot change the routing tables of routers.

Single path, free routing. We can choose the path from P_q to P_r, but a single route must be used for all data originating from P_q and targeting P_r. This policy corresponds to protocols allowing us to choose the route for any of the data transfer, and to reserve some bandwidth on the chosen routes. Although current network protocols do not provide this feature, bandwidth reservation, and more generally resource reservation in Grid network, is the subject of a wide literature, and will probably be available in future computing platforms [45].

Multiple paths. Data from P_q to P_r may be split along several routes taking different paths. This corresponds to the uttermost flexible case where we can simultaneously reserve several routes and bandwidth fractions for concurrent transfers.

The three routing policies allow us to model a wide range of practical situations, current and future. The techniques exposed in Section 4.3 enable us to deal with any of these models and even with combinations of them.

In the case of single path policies, $\sigma(F_{i,j})$ is the set of the links constituting the path. In the case of multiple paths, $\sigma(F_{i,j})$ is a weighted set of paths $\{(w_\alpha, P_a)\}$: for example $\sigma(F_{7,8}) = \{(0.1, P_1 \to P_3), (0.9, P_1 \to P_2 \to P_3)\}$ means that 10% of the file $F_{7,8}$ go directly from P_1 to P_3 and 90% are transfered through P_2.

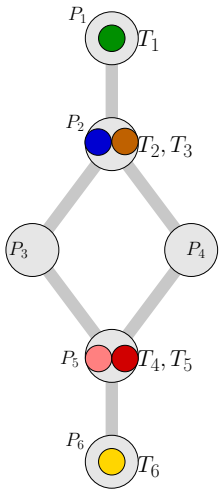
Figure 4.1 gives, for each routing policy, an example of allocation of the application graph of Figure 4.2(b). The task mapping is always the same: $\sigma(T_1) = P_1$, $\sigma(T_2) = \sigma(T_3) = P_2$, $\sigma(T_4) = \sigma(T_5) = P_5$ and $\sigma(T_6) = P_6$. In Figure 4.1(b), the path for any transfer is fixed; in particular, all data from P_2 targeting P_5 must pass through P_3. In Figure 4.1(c), we assume a free routing policy: we can choose to route some files via P_3 and some others via P_4. Finally, in Figure 4.1(d), we are allowed to use multiple paths to route a single file, which is done for file $F_{3,5}$.

4.2.3 Upper bound on the achievable throughput

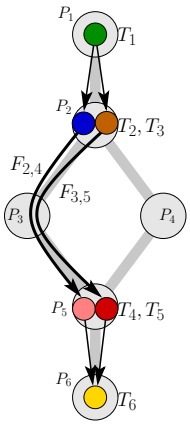
We first derive a tight upper bound on the throughput of any schedule, as it was defined in Section 3.3. We are only interested in very specific schedules, consisting of only one allocation. We now show how to compute an upper bound on the achievable throughput for a given allocation. We later show that this bound is tight.

First, we consider the time spent by each resource on one instance of a given allocation σ. In other words, we consider the time spent by each resource for processing a single copy of our workflow under allocation σ.

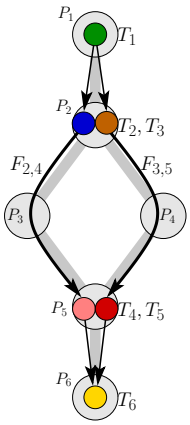
- The computation time spent by a processor P_q for processing a single instance is: $t_q^{\text{comp}} = \sum_{i,\sigma(T_i)=P_q} w_{i,q}$.
- The total amount of data carried by a communication link $P_q \to P_r$ for a single instance is $d_{q,r} = \sum_{(i,j),P_q \to P_r \in \sigma(F_{i,j})} data_{i,j}$ for single-path policies, and $d_{q,r} = \sum_{F_{i,j}} \sum_{\substack{(w_a,P_a)\in\sigma(F_{i,j}) \\ P_q \to P_r \in P_a}} w_a \times data_{i,j}$ for the multiple-paths policy. This allows us to compute the time spent by each link, and each network interface, on this instance:
  - on link $P_q \to P_r$: $t_{q,r} = d_{q,r}/bw_{q,r}$;



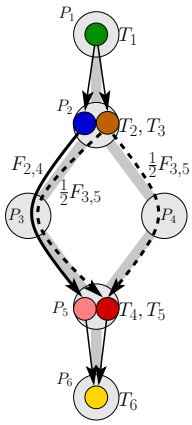
(a) Task allocation.



(b) Single path, fixed routing.



(c) Single path, free routing.



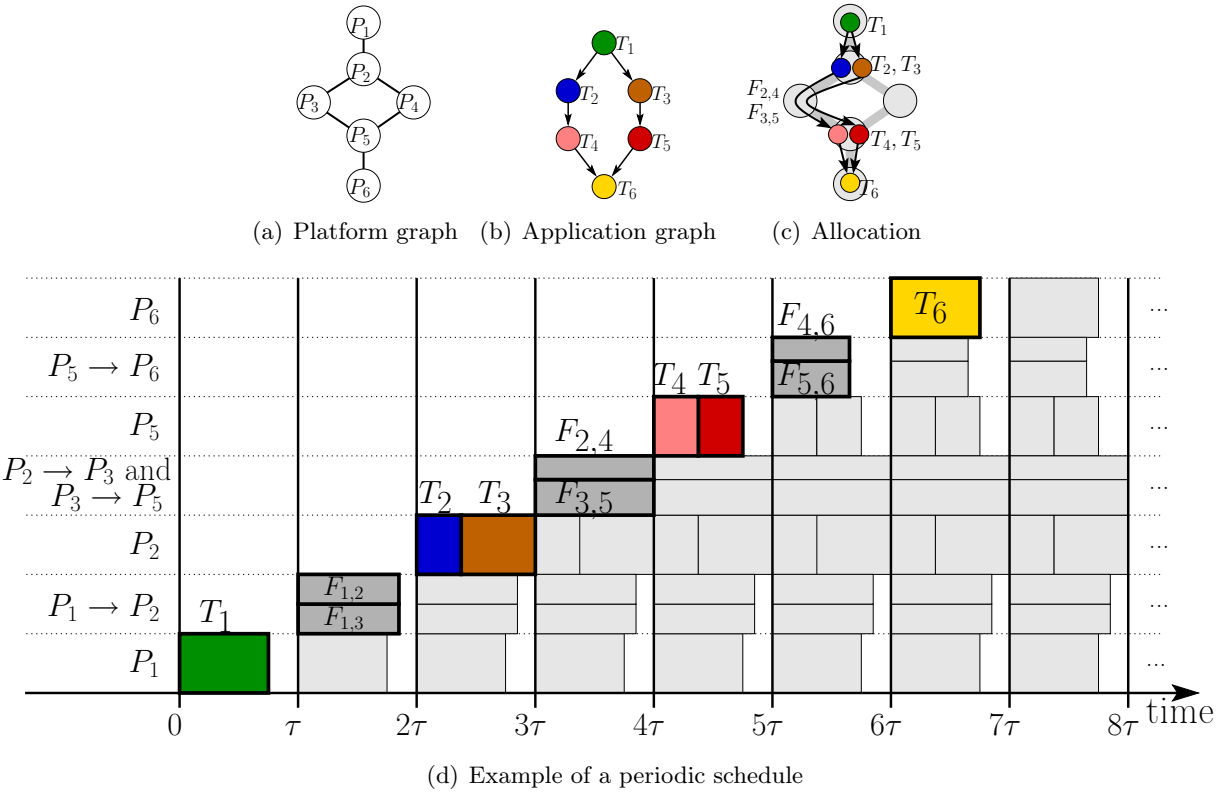
(d) Multiple paths.

Figure 4.1: Allocation examples for various routing policies.

– on P_q outgoing interface: $t_q^{\text{out}} = \sum_r d_{q,r}/bw_q^{\text{out}}$;

– on P_q incoming interface: $t_q^{\text{in}} = \sum_r d_{r,q}/bw_q^{\text{in}}$.

We can now compute the maximum time $\mathcal{T}$ spent by any resource for the processing of one instance: $\mathcal{T} = \max\left\{\max_{P_q}\{t_q^{\text{comp}}, t_q^{\text{out}}, t_q^{\text{in}}\}, \max_{P_q \to P_r} t_{q,r}\right\}$. This gives us an upper bound on the achievable throughput: $\rho \leq \rho_{\max} = 1/\mathcal{T}$. Indeed, as there is at least one resource which spends a time $\mathcal{T}$ to process its share of a single instance, the throughput cannot be greater than 1 instance per $\mathcal{T}$ units of time. We now show that this upper bound is achievable in practice, i.e., that there exists a schedule with throughput $\rho_{\max}$. In the following, we call "throughput of an allocation" the optimal throughput $\rho_{\max}$ of this allocation.



(a) Platform graph (b) Application graph (c) Allocation

(d) Example of a periodic schedule

Figure 4.2: Example of periodic schedule. Only the first instance is represented with task and file labels.

The upper bound is achievable. Here, we will only explain on an example how one can build a periodic schedule achieving the throughput $\rho_{\max}$. Indeed, we are not interested here in giving a formal definition of periodic schedules, nor to formally define and prove schedules which achieve the desired throughput, as this goes far beyond the scope of this chapter. The construction of such schedules, for applications modeled by DAGs, was introduced in [15], and a fully comprehensive proof can be found in [17].

Figure 4.2 illustrates how to build a periodic schedule of period $\mathcal{T}$ for the workflow described on Figure 4.2(b), on the platform of Figure 4.2(a), using the allocation of Figure 4.2(c). Once

the schedule has reached its steady state, that is after $6\mathcal{T}$ in the example, during each period, each processor computes one instance of each task assigned to it. More precisely, in steady state, during period k ($k \geq 6$), that is during time-interval $[k\mathcal{T}; (k+1)\mathcal{T}]$, the following operations happens:

- P_1 computes task T_1 of instance k,
- P_1 sends $F_{1,2}$ and $F_{1,3}$ of instance $k-1$ to P_2,
- P_2 processes T_2 and T_3 of instance $k-2$,
- P_2 sends $F_{2,4}$ and $F_{3,5}$ of instance $k-3$ to P_5 (via P_3),
- P_4 processes tasks T_4 and T_5 of instance $k-4$,
- P_5 sends $F_{4,6}$ and $F_{5,6}$ of instance $k-5$ to P_6,
- P_6 processes task T_6 of instance $k-6$.

One instance is thus completed after each period, achieving a throughput of $1/\mathcal{T}$.

4.2.4 NP-completeness of throughput optimization

We now formally define the decision problem associated to the problem of maximizing the throughput.

Definition 4.1 (DAG-Single-Alloc)**.** *Given a directed acyclic application graph G_A, a platform graph G_P, and a bound B, is there an allocation with throughput $\rho \geq B$?*

Theorem 4.1. *DAG-Single-Alloc is NP-complete for all routing policies.*

Proof. We first have to prove that the problem belongs to NP, that is that we can check in polynomial time that the throughput of a given allocation is greater than or equal to B. Thanks to the previous section, we know that this check can be made through the evaluation of a simple formula; DAG-Single-Alloc is thus in NP.

To prove that DAG-Single-Alloc is NP-complete, we use a reduction from the Minimum Multiprocessor Scheduling, known to be NP-complete [46]. Consider an instance $\mathcal{I}_1$ of Multiprocessor Scheduling, that is a set of n independent tasks $T_{i,1\leq i\leq n}$ and a set of m processors $P_{u,1\leq u\leq m}$, where task i takes time $t(i,u)$ to be processed on processor P_u. The problem is to find a schedule with total execution time less than a given bound T. We construct a very similar instance of DAG-Single-Alloc:

- The application DAG is a simple fork, made of all tasks T_i plus a task T_0, root of the fork: for each $1 \leq i \leq n$, there is an edge $F_{0,i}$, with $data_{0,i} = 0$.
- The platform consists of the same set of processors than $\mathcal{I}_1$, connected with a complete network where all bandwidths are equal to 1. The time needed to process task T_i on processor P_u is $w_{i,u} = t(i,u)$ for each $1 \leq i \leq n$, and $w_{0,u} = 0$.

Note that communications need not being taken into account during performance evaluation, since all data sizes are null. Thus, this reduction applies to any routing policy. The throughput of an allocation is directly related to the total execution time of the set of tasks: an allocation has throughput ρ if and only if it completes all the tasks in time $1/\rho$. Thus finding a schedule with completion time less than T is equivalent to finding an allocation with throughput greater than $1/T$. ■

4.3 Mixed linear program formulation for optimal allocations

In this section, we present a mixed linear program formulation that allows to find optimal allocations with respect to the total throughput.

4.3.1 Single path, fixed routing

In this section, we assume that the path to be used to transfer data from a processor P_q to a processor P_r is determined in advance; we have thus no freedom on its choice. We then denote by $P_q \rightsquigarrow P_r$ the set of edges of E_P which are used by this path.

Our linear programming formulation makes use of both integer and rational variables. The resulting optimization problem, although NP-complete, is solvable by specialized softwares (see Section 4.5 about simulations). The integer variables can take 0 or 1 value. The only integer variables are the following:

- y's variables which characterize where each task is processed: $y_q^k = 1$ if and only if task T_k is processed on processor P_q;
- x's variables which characterize the mapping of file transfers: $x_{q,r}^{k,l} = 1$ if and only if file $F_{k,l}$ is transfered using path $P_q \rightsquigarrow P_r$; note that we may well have $x_{q,q}^{k,l} = 1$ if processor P_q executes both tasks T_k and T_l.

Obviously, these two sets of variables are related. In particular, for any allocation, $x_{q,r}^{k,l} = y_q^k \times y_r^l$. This redundancy allows us to write linear constraints.

$$
\left\{
\begin{array}{ll}
\multicolumn{2}{l}{\text{MINIMIZE } \mathcal{T} \text{ UNDER THE CONSTRAINTS}} \\
\text{(4.1a)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, \quad x_{q,r}^{k,l} \in \{0,1\}, \quad y_q^k \in \{0,1\} \\
\text{(4.1b)} & \forall T_k, \quad \sum_{P_q} y_q^k = 1 \\
\text{(4.1c)} & \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, \quad x_{q,r}^{k,l} \leq y_q^k \\
\text{(4.1d)} & \forall T_l, \forall F_{k,l}, \forall P_r, \quad y_r^k + \sum_{P_q \rightsquigarrow P_r} x_{q,r}^{k,l} \geq y_r^l \\
\text{(4.1e)} & \forall P_q, \quad \sum_{T_k} y_q^k w_{q,k} \leq \mathcal{T} \\
\text{(4.1f)} & \forall P_q \to P_r, \quad d_{q,r} = \displaystyle\sum_{\substack{P_s \rightsquigarrow P_t \text{ with} \\ P_q \to P_r \in P_s \rightsquigarrow P_t}} \sum_{F_{k,l}} x_{s,t}^{k,l} data_{k,l} \\
\text{(4.1g)} & \forall P_q \to P_r, \quad \frac{d_{q,r}}{bw_{q,r}} \leq \mathcal{T} \\
\text{(4.1h)} & \forall P_q \quad \sum_{P_q \to P_r \in E_P} \frac{d_{q,r}}{bw_q^{\text{out}}} \leq \mathcal{T} \\
\text{(4.1i)} & \forall P_r \quad \sum_{P_q \to P_r \in E_P} \frac{d_{q,r}}{bw_q^{\text{in}}} \leq \mathcal{T}
\end{array}
\right. \tag{4.1}
$$

Linear Program (4.1) expresses the optimization problem for the fixed-routing policy. The objective function is to minimize the maximum time $\mathcal{T}$ spent by all resources, in order to maximize the throughput $1/\mathcal{T}$. The intuition behind the linear program is the following:

- Constraints (4.1a) define the domain of each variable: x, y lie in $\{0,1\}$, while $\mathcal{T}$ is rational.
- Constraint (4.1b) ensures that each task is processed exactly once.

- Constraint (4.1c) asserts that a processor can send the output file of a task only if it processes the corresponding task.
- Constraint (4.1d) asserts that the processor computing a task holds all necessary input data: for each predecessor task, it either received the data from that task or computed it.
- Constraint (4.1e) ensures that the computing time of a processor is no larger that $\mathcal{T}$.
- In Constraint (4.1f), we compute the amount of data carried by a given link, and the following constraints ((4.1g),(4.1h),(4.1i)) ensure that the time spent on each link or interface is not larger than $\mathcal{T}$, with a formulation similar to that of Section 4.2.3.

We denote $\rho^* = 1/\mathcal{T}_{\text{opt}}$, where $\mathcal{T}_{\text{opt}}$ is the value of $\mathcal{T}$ in any optimal solution of Linear Program (4.1). The following theorem states that ρ^* is the maximum achievable throughput.

Theorem 4.2. *An optimal solution of Linear Program (4.1) describes an allocation with maximal throughput for the fixed routing policy.*

Proof. We first prove that for any allocation of throughput ρ, described by x's and y's variables, $(x, y, \mathcal{T} = 1/\rho)$ satisfies the constraints of the linear program.

- All tasks of the workflow are processed exactly once, thus Constraint (4.1b) is verified.
- In any allocation, $x_{q,r}^{k,l} = y_q^k \times y_r^l$, thus Constraint (4.1c) is verified.
- If $y_r^l = 1$, that is if P_r processes T_l, then P_r must own all files $F_{k,l}$. It can either have it because it also processed T_k (in this case $y_r^k = 1$) or because it received it from some processor P_q (and then $x_{q,r}^{k,l} = 1$). In both cases, Constraint (4.1d) is satisfied.
- As the allocation has throughput ρ, it means that the occupation time of each processor, each link, and each network interface is at most $1/\rho$. This is precisely what is stated by Constraints (4.1e), (4.1g), (4.1h) and (4.1i). Thus, these constraints are satisfied.

As all allocations satisfy the constraint of the linear program, and ρ^* is the maximal value of the throughput under these constraints, then all allocations have a throughput smaller than, or equal to, ρ^*.

We now prove that any solution of the linear program represents an allocation. We have to verify that all tasks are performed, all input data needed to process a task are correctly sent to the corresponding processor, and that the allocation has the expected throughput $1/\mathcal{T}$.

- All tasks are processed exactly once due to constraint (4.1b).
- Thanks to Constraint (4.1c), a processor is allowed to send a file $F_{k,l}$ only if it processed T_k.
- Thanks to Constraint (4.1d), a task T_l with predecessor T_k is processed on P_r only if P_r either processed T_k, or received $F_{k,l}$ from some processor P_q.
- Thanks to Constraints (4.1e), (4.1g), (4.1h) and (4.1i), we know that the maximum utilization time of any resource (processor, link or network interface) is at least equal to $\mathcal{T}$, the corresponding allocation has a throughput at most $1/\mathcal{T}$.

Thus, any solution of the linear program describes a valid allocation. In particular, there is an allocation with throughput ρ^*. ■

4.3.2 Single path, free routing

We now move to the free routing setting. The transfer of a given file between two processors can take any path between these processors in the platform graph. We introduce a new set of variables to take this into account. For any file $F_{k,l}$ and link $P_i \to P_j$, $f_{i,j}^{k,l}$ is an integer value, with value 0 or 1: $f_{i,j}^{k,l} = 1$ if and only if the transfer of file $F_{k,l}$ between the processor processing T_k to the one processing T_l takes the link $P_i \to P_j$. Using these new variables, we transform the previous linear program to take into account the free routing policy. The new program, Linear Program (4.2) has exactly the same constraints than Linear Program (4.1) except for the following:

1. the new variables are introduced (Constraint (4.2a));
2. the computation of the amount of data in Constraint (4.1f) is modified into Constraint (4.2f) to take into account the new definition of the routes;
3. the new set of constraints (4.2j) ensures that a flow of value 1 is defined by the variables $f^{k,l}$ from the processor executing T_k to the one executing T_k.

$$
\begin{cases}
\text{MINIMIZE } \mathcal{T} \text{ UNDER THE CONSTRAINTS} \\
\text{(4.2a)} \quad \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, \\
\qquad x_{q,r}^{k,l} \in \{0,1\},\ y_q^k \in \{0,1\},\ f_{i,j}^{k,l} \in \{0,1\} \\
\text{(4.2f)} \quad \forall P_q \to P_r,\ \ d_{q,r} = \sum_{F_{k,l}} f_{i,j}^{k,l}\, data_{k,l} \\
\text{(4.2j)} \quad \forall P_q, \forall F_{k,l},\ \ \sum_{P_q \to P_r} f_{q,r}^{k,l} - \sum_{P_{r'} \to P_q} f_{r',q}^{k,l} \\
\qquad\qquad = \sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} \\
\text{AND (4.1b), (4.1c), (4.1d), (4.1e), (4.1g), (4.1h), (4.1i)}
\end{cases}
\tag{4.2}
$$

In the following lemma, we clarify the role of the f variables.

Lemma 4.1. *Given a file $F_{k,l}$, the following two properties are equivalent*

(i) $\forall P_q,\quad \displaystyle\sum_{P_q \to P_r} f_{q,r}^{k,l} - \sum_{P_{r'} \to P_q} f_{r',q}^{k,l} = \begin{cases} 1 & \text{if } P_q = P_{\text{prod}} \\ -1 & \text{if } P_q = P_{\text{cons}} \\ 0 & \text{otherwise} \end{cases}$

(ii) the set of links $P_q \to P_r$ such that $f_{q,r}^{k,l} = 1$ defines a route from P_{prod} to P_{cons}.

Proof. The result is straightforward since Property (i) is a simple conservation law of f quantities. Note that this route may include cycles. These cycles do not change the fact that the links can be used to ship the files from P_{prod} to P_{cons}, but the time needed for the transportation is artificially increased. That is why in our experiments these cycles are sought and deleted to keep routes as short as possible. ■

The following theorem states that the linear program computes an allocation with optimal throughput: again, we denote $\rho^* = 1/\mathcal{T}_{\text{opt}}$, where $\mathcal{T}_{\text{opt}}$ is the value of $\mathcal{T}$ in any optimal solution of this linear program.

Theorem 4.3. *An optimal solution of Linear Program (4.2) describes an allocation with optimal throughput for the free routing policy.*

Proof. Similarly to the proof of Theorem 4.2, we first consider an allocation, define the x, y and f variables corresponding to this allocation, and prove that they satisfy the constraints of the linear program.

- Since $f_{i,j}^{k,l}$ describes if transfer $F_{k,l}$ uses link $P_i \to P_j$, all constraints except (4.2j) are satisfied by the same justifications as in Theorem 4.2.
- In any allocation, file $F_{k,l}$ must be routed from the processor executing T_k to the processor executing T_l (provided that these tasks are excepted by different processors). Thus, f define a route between those two processors. Then, we note that

$$\sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} = \begin{cases} 1 & \text{if } P_q \text{ executes } T_k \text{ and not } T_l \\ -1 & \text{if } P_q \text{ executes } T_l \text{ and not } T_k \\ 0 & \text{otherwise} \end{cases}$$

  If T_k and T_k are executes on the same processor, all corresponding f and x variables are equal to 0. Thus, thanks to Lemma 4.1, all Constraints (4.2j) are verified.

We now prove that any solution of Linear Program (4.2) defines a valid allocation with throughput ρ under the Free routing policy.

- As in the proof of Theorem 4.2, we can prove that x and y variables define an allocation with throughput ρ.
- As above, we note that for a given file $F_{k,l}$

$$\sum_{P_t} x_{q,t}^{k,l} - \sum_{P_s} x_{s,q}^{k,l} = \begin{cases} 1 & \text{if } P_q \text{ executes } T_k \text{ and not } T_l \\ -1 & \text{if } P_q \text{ executes } T_l \text{ and not } T_k \\ 0 & \text{otherwise} \end{cases}$$

  If tasks T_k and T_l are not executed on the same processors, we know thanks to Lemma 4.1 that the f variables define a route from the processor executing T_k to the one executing T_l.

Thus, any solution of the linear program describes a valid allocation. In particular, there is an allocation with throughput ρ^*. ■

4.3.3 Multiple paths

Finally, we present our linear programming formulation for the most flexible case, the multiple-paths routing: any transfer may now be split into several routes in order to increase its throughput. The approach is extremely similar to the one used for the single route, free routing policy: we use the same set of f variables to define a flow from processors producing files to processors consuming them. The only difference is that we no longer restrict f to integer values: by using rational variables in $[0; 1]$, we allow each flow to use several concurrent routes. Theorem 4.4 expresses the optimality of the allocation found by the linear program. Its proof is very similar to the proof of Theorem 4.3.

$$\begin{cases} \text{MINIMIZE } \mathcal{T} \text{ UNDER THE CONSTRAINTS} \\ \text{(4.3a)} \quad \forall F_{k,l}, \forall P_q \rightsquigarrow P_r, \\ \qquad x_{q,r}^{k,l} \in \{0,1\},\ y_q^k \in \{0,1\},\ f_{i,j}^{k,l} \in [0;1] \\ \text{AND (4.1b), (4.1c), (4.1d), (4.1e), (4.2f), (4.1g), (4.1h), (4.1i), (4.2j)} \end{cases} \tag{4.3}$$

Theorem 4.4. *An optimal solution of Linear Program (4.3) describes an allocation with optimal throughput for the multiple paths policy.*

Proof. Consider the subgraph of the platform graph comprising only the links $P_q \to P_r$ such that $f_{q,r}^{k,l} \neq 0$. We construct the set of weighted routes by the following iterative process. We extract a route r from P_{prod} to P_{cons} from this graph (such a route exists thanks to the conservation law). We then compute the minimum weight w of the links in route r. Route r is added with weight w to the set of weighted routes, and the subgraph is pruned as followed: w is subtracted from the value of $f^{k,l}$ of all links included in route r, and links whose $f^{k,l}$ value becomes null are removed from the subgraph. We can prove that Property (i) still holds with value $1 - w$ instead of 1. We continue the process until there is no more link in the subgraph. ■

4.4 Heuristics

Due to the exponential complexity of the algorithms used to solve mixed linear programs, using the formulation given in the previous section is unrealistic for large problems. One could think to use the algorithm described in [15] for finding the optimal solution using several allocations, selecting only the most significant one. However, this method leads to very large linear programs: the number of variables may exceed 500,000 on problems with around 20 tasks and 20 processors, leading to an excessive memory consumption. Thus, we present in this section several faster heuristics, which are more adapted to large problems.

4.4.1 Greedy mapping policies

In this section, we propose greedy strategies to find an allocation of task graphs on processors. Greedy algorithms are fast, easy to implement, and often effective to find reasonable solutions.

Simple greedy. This heuristic is described by Algorithm 2. We first compute the weight of a task T_k as its maximum execution time over all processors. Then we consider the task with maximum weight, and allocate it to a processor so that the updated computation time of the processor is minimum. As we focus on steady state, we do not consider dependencies and only try to minimize the processor occupation time. (Dependencies are taken care of when building the schedule from the steady-state characterization [19].)

Refined greedy. The previous heuristic attempts to balance the computing workload on processors, but do not take communications into account. We try to refine the search of an allocation in this heuristic, inspired from the classical HEFT algorithm for task graph scheduling [79]. Again, dependencies are not taken into account since we focus on steady state (see the remark on SIMPLE_GREEDY). Algorithm 3 describes this heuristic.

Algorithm 2: **Simple_greedy**(G_P, G_A)

```
foreach T_k do weight[T_k] ← max_{P_i} w_{i,k};
foreach P_i do processing_time[P_i] ← 0;
foreach T_k in decreasing order of weight do
    Find P_i such that processing_time[P_i] + w_{i,k} is minimized;
    processing_time[P_i] ← processing_time[P_i] + w_{i,k};
    mapping[T_k] ← P_i;
return mapping;
```

Algorithm 3: **Refined_greedy**(G_P, G_A)

```
avg_comp_time[T_k] ← average computation time of T_k over all processors;
avg_comm_time[F_{k,l}] ← average communication time of F_{k,l} over all links;
foreach source T_k of the task graph do
    weight[T_k] ← 0;
foreach T_l in topological order do
    weight[T_l] ← max_{F_{k,l}} (weight[T_k] + avg_comm_time[F_{k,l}]) + avg_comp_time[T_l];
foreach T_k in decreasing order of weight do
    Find P_i such that the maximum of occupation time over all resources is minimized;
    mapping[T_k] ← P_i;
    Update the occupation time of all involved resources (P_i and communication links);
return mapping;
```

4.4.2 Rounding of the linear program

Since our problem is expressed as a mixed linear program, a natural way to find a solution is to relax the linear program to solve it over rational numbers, and then to round-off the rational solution into an integer one. Several different approaches exist for the rounding-off of rational solutions. We present two of them: a greedy rounding and a randomized one. Both variants are described in Algorithm 4.

Algorithm 4: **RLP**(G_P, G_A)

```
Constraints ← initial set of constraints, given in Linear Program (4.1);
for m = 1 to n do
    Solve over the rationals the linear program associated to Constraints;
    if using RLP_max then
        Find the maximum of the y_i^k's over all T_k's and P_i's, such that y_i^k has not yet
        been set;
    else if using RLP_rand then
        Randomly choose some task T_k which has not yet been mapped;
        Randomly choose a processor P_i, using probability y_j^k for P_j ;
    Constraints ← Constraints ∪ {y_i^k = 1};
```

In the first variant (RLP_MAX), at each step, we search among the y_i^k's (which have not yet been set) for the one with the largest value. This y_i^k is then set to 1 in the Linear Program. After n steps, each task has been mapped to a processor, which defines a whole allocation.

In the second variant (RLP_RAND), we make use of a randomized rounding, which is known to sometimes lead to very efficient solutions [36]. At each step, we randomly select a task that has not yet been mapped. Then we randomly choose the processor that will process this task using a probability distribution defined by the y_i^k: each processor P_i has probability y_i^k to be chosen for the processing.

4.4.3 An involved strategy to delegate computations

In this section, we present an iterative strategy to build an efficient allocation. Contrarily to the previous heuristics, this algorithm is not based on the linear program formulation. This method, called DELEGATE, consists in iteratively refining an allocation by moving some work from a highly loaded processor to a less loaded one. In the beginning, all tasks are mapped to the source processor P_{source}. Then, we select one task and some of its neighbors and *delegate* them to another processor. This refinement procedure is repeated as long as the throughput can be improved, as described in Algorithm 5.

Algorithm 5: **DELEGATE**(G_P, G_A, *depth*)

```
foreach T_k do current_mapping[T_k] ← P_source;
current_value ← evaluate(current_mapping);
continue ← TRUE;
while continue do
    best_value ← 0;
    foreach T_k do
        foreach P_i such that current_mapping[T_k] ≠ P_i do
            forall connected neighborhood S of T_k do
                mapping ← move (current_mapping, S, P_i);
                mapping ← refine_move (mapping, S, P_i);
                value ← evaluate(mapping);
                if (value > best_value) then
                    (best_value,best_mapping) ← (value,mapping);
    if (best_value > current_value) then
        (current_value,current_mapping) ← (best_value,best_mapping);
        continue ← TRUE;
    else continue ← FALSE;
return current_mapping;
```

At each step, we consider a candidate move (T_k,P_i), i.e., delegating task T_k to processor P_i (assuming that T_k is not already mapped to P_i). Delegating a single task T_k to another processor may not be interesting because of communications involving this task. Thus, we look for a cluster of tasks containing T_k that it would be beneficial to delegate to P_i.

We define a neighborhood of T_k as 1) a connected set of tasks, 2) which contains T_k, and 3) which only contains tasks which are at most at a distance *depth* from T_k in the task graph. *depth* is a parameter of the algorithm. In practice we set the value of *depth* to 2. We will see in Section 4.5 that this value is large enough for our purpose.

We test all neighborhoods, trying in turn to map each of them on processor P_i. This is done through the **move** function. We then select the best move among all neighborhoods of all tasks.

If this best move induces an improvement in performance, we perform the move. Otherwise, we end the overall process.

Evaluation metric. This algorithm strongly depends on the evaluation function, which is used both to identify the best move, and to decide whether the overall performance is improved. We have several possible choices for this evaluation function:

- Obviously, we could use the throughput of an allocation as a measure of its quality. We can compute the throughput as described in Section 4.2.3: the total throughput is the inverse of the maximum occupation time of any resource. It can similarly be computed with
$$\rho = \min\left\{\min_{P_q}\left\{\frac{1}{t_q^{\text{comp}}}, \frac{1}{t_q^{\text{out}}}, \frac{1}{t_q^{\text{in}}}\right\}, \min_{P_q \to P_r} \frac{1}{t_{q,r}}\right\}.$$
Using the global throughput allows us to ensure that the overall performance is improved at each step, but may lead to sub-optimal scenarios: when two processors are evenly loaded, we can only decrease the occupation time of a single processor at each algorithm step. Two successive moves are thus required for the overall throughput to decrease. This cannot be done with this evaluation function the way we designed Algorithm 5.
- To overcome the issue of using the throughput metric, we rather use a different way to compare two allocations, thanks to the lexicographical order. Instead of computing a single value for each allocation, we sort all resource occupation times by decreasing order, and use the lexicographical order to compare two allocations. The underlying idea is to first minimize the occupation of the most used resource —the one defining the throughput— and then the occupation of the other resources.

Further improvements. Algorithm 5 can be improved in several ways:

- To keep computation time low, we have to set parameters *depth* to small values. However, this prevents large subset of tasks to be simultaneously delegated to a same processor. Thus, we introduce a function, **refine_move**, which enlarges the subset of tasks to delegate to a processor. It greedily considers in an arbitrary order any neighbor task of the tasks currently in the subset S, and add this task to S if this leads to a better allocation.
- The search for an allocation continues until Algorithm 5 cannot improve the allocation any further. When using the lexicographical order, however, the number of iterations might be large before no more local improvement is possible. To keep a low execution time, we could bound the maximum number of iterations.

In the following experiments, we always use the **refine_move** improvement, but never bound the number of iterations.

4.4.4 A neighborhood-centric strategy

In this section, we expose a simpler strategy called NEIGHBORHOOD, based on the evaluation of the mapping of a task T_k and its neighbors, i.e., any task T_l such the file $F_{k,l}$ (or the $F_{l,k}$) exists. For any task, we initially evaluate the cost of the mapping of it and its neighborhood on an idle platform to define its priority: the higher this cost, the higher its priority. For any pair (T_k, P_i), we consider all the neighbors of T_k in an arbitrary order, temporarily mapping them on the processor ensuring the best partial mapping. To compare two partial mappings, we first compare the occupation times of the most used resources, or those of the second most

used resources in case of tie, and so on. The priority of T_k is then defined as the average of the inverse of the throughputs of the partial mappings over all processors P_i.

After this evaluation, we consider all tasks in the decreasing order of their priority. At each step, we consider a candidate processor P_i for the task T_k. Given this processor, we consider the neighbors of T_k in an arbitrary order, temporarily mapping them on the processor which minimizes the occupation time (the exact comparison of two partial mappings is done as in the previous paragraph). We finally map T_k on the processor, which allows the best partial mapping. The complete procedure is described in Algorithm 6.

Algorithm 6: NEIGHBORHOOD(G_P, G_A)

foreach T_k **do**
  foreach P_i **do**
    foreach P_i **do** $time[P_i] \leftarrow 0$;
    foreach $P_i \rightarrow P_j$ **do** $time[P_i \rightarrow P_j] \leftarrow 0$;
    $time[P_i] \leftarrow w_{i,k}$;
    foreach *neighbor* T_l *of* T_k **do**
      foreach P_j **do**
        $time_j \leftarrow time$;
        $time_j[P_j] \leftarrow time_j[P_j] + w_{j,l}$;
        add to $time_j$ the cost of the transfer of $F_{k,l}$ (or $F_{l,k}$) from P_i to P_j ;
      $time \leftarrow \min_{P_j}(time_j)$;
    $time'_i \leftarrow time$;
  $priority[T_k] \leftarrow \text{average}_{P_i}(time'_i)$;
foreach P_i **do** $time[P_i] \leftarrow 0$;
foreach $P_i \rightarrow P_j$ **do** $time[P_i \rightarrow P_j] \leftarrow 0$;
foreach T_k *in decreasing order of priority* **do**
  foreach P_i **do**
    $time_i \leftarrow time$;
    $time_i[P_i] \leftarrow time_i[P_i] + w_{i,k}$;
    foreach *neighbor* T_l *of* T_k **do**
      if $mapping[T_l]$ *is undefined* **then**
        foreach P_j **do**
          $time'_j \leftarrow time_i$;
          $time'_j[P_j] \leftarrow time'_j[P_j] + w_{j,l}$;
          add to $time'_j$ the cost of the transfer of $F_{k,l}$ (or $F_{l,k}$) from P_i to P_j ;
        $time_i \leftarrow \min_{P_j}(time'_j)$;
      else
        add to $time_i$ the cost of the transfer of $F_{k,l}$ (or $F_{l,k}$) from P_i to $P_{mapping[T_l]}$;
  $time \leftarrow \min_{P_i}(time_i)$;
  $mapping[T_k] \leftarrow \text{argmin}_{P_i}(time_i)$;
return *mapping*;

4.5 Performance evaluation

In this section, we present the simulations performed to study the performance of our strategies. Simulations allow us to test different heuristics on the very same scenarios, and also to consider far more scenarios than real experiments would. We can even test scenarios that would be quite hard to run real experiments with. This is especially true for the flexible or multiple-path routing policies. Our simulations consist here in computing the throughput obtained by a given set of heuristics on a given platform, for some application graphs. We also study another metric: the latency of the heuristics, that is the time between the beginning and the end of the processing of one input data set. A large latency may lead to a bad quality of service in the case of an interactive workflow (e.g., in image processing), and to a huge amount of temporary files. This is why we intend to keep the latency low for all input data sets.

4.5.1 Reference heuristics

In order to assess the quality and usefulness of our strategies, we compare them against three classical task-graph scheduling heuristics. These heuristics (HEFT, Data-Parallel and Clustering) are dynamic strategies: they allocate resources to tasks in the order of their arrival.

HEFT. This heuristic builds up a schedule by applying the classical Heterogeneous Earliest Finish Time [79] strategy to a collection of a given number (usually $1,000$) of instances of the original task graph.

Pure data-parallelism. We also compare our approach to a pure data-parallelism strategy: in this case, all tasks of a given instance are processed sequentially on a given processor, as detailed in the introduction.

Clustering. Another reference heuristic is the clustering method presented by Sarkar in [71]. This method gathers tasks into clusters before dispatching these clusters on the available processors, trying to minimize the overall communication cost.

Multi-allocations upper bound. In addition to the previous classical heuristics, we also study the performance when mixing control- and data-parallelism. This approach uses concurrent allocations to reach the optimal throughput of the platform, as is explained in detail in [17]. Rather than using the complex algorithm described in that chapter for task graphs with bounded dependencies, we use an upper bound on the throughput based on this study, which consists of a simple linear program close to the one described in this chapter, and solved over the rational numbers. This bound is tight when the task graph is an in- or out-tree, but may not take all dependencies into account otherwise. This upper bound, however, has proved to be a very good comparison basis in the following, and is used as a reference to assess the quality of other heuristics. No latency can be derived from this bound on the throughput, since no real schedule is constructed.

4.5.2 Simulation settings

All algorithms are simulated using the SimGrid framework [31]. The number N of instances to process is set to a large number (between 100 and 1000). For all steady-state strategies (MLP, Simple_Greedy, Refined_Greedy, RLP_max, RLP_rand, and Delegate), we run

the corresponding algorithm to build the allocation; then we construct the schedule corresponding to the steady-state allocation for the N instances. Then, each schedule is executed in the SimGrid simulator [31]. We compute the experimental throughput as the ratio between the total completion time and the number N of instances. For steady-state heuristics, we also compute a theoretical throughput, based on the study of Section 4.2.3. The theoretical and experimental throughputs may differ due to the slight differences between our multiport model and the SimGrid network model. Nevertheless, they are quite close in our experiments. Table 4.1 gives the average error (and its standard deviation) between theoretical and experimental throughputs for each algorithm. The more communication-aware the heuristics, the smaller the error.

Algorithm	Average error	Standard deviation
MLP	3%	3%
SIMPLE_GREEDY	8%	11%
REFINED_GREEDY	5%	6%
RLP_MAX	8%	12%
RLP_RAND	16%	28%
DELEGATE	2%	2%
NEIGHBORHOOD	6%	12%

Table 4.1: Average error (and its standard deviation) between theoretical and experimental throughputs for each algorithm.

We perform two sets of simulations. First, we compare all algorithms on rather small problems (up to 12 tasks in the task graphs); we have 135 platform/application scenarios in this set. Then, we compare the heuristics on larger simulation settings, with task graphs including up to 47 tasks; this set comprises 445 scenarios. We exclude the MLP algorithm from the latter set of problems because of its prohibitive running time on the larger task graphs (several days).

Platforms. We use several platforms representing existing computing Grids. The descriptions of the platforms were obtained through the SimGrid simulator repository [31]:

- DAS-3, the Dutch Grid infrastructure,
- Egee, a large-scale European multi-disciplinary Grid infrastructure, gathering more than 68,000 CPUs,
- Grid5000, a French research Grid with targets 5000 processors,
- GridPP, the UK Grid infrastructure for particle physics.

Most of the time, users do not have access to a whole Grid but to a limited subset of a Grid, usually through a reservation. To simulate this behavior, a subset of the available processors is first randomly selected for each platform/application scenario, and then used by all heuristics. It is composed of around 10 processors for the small problems, and between 40 and 70 processors for larger problems. To evaluate our fixed-routing strategies, we pre-compute a shortest-path route between any pair of processors, which is used as the compulsory route.

Applications. Several workflows are used to assess the quality of our strategies, with a number of tasks between 8 and 12, and up to 47 tasks otherwise:

- pipeAlign [67], a protein family analysis tool,
- several random task graphs generated by the TGFF generator [40],
- several random task graphs generated by the DagGen generator [77].

In order to evaluate the impact of communications on the quality of the result, we artificially modify the applications' communication-to-computation ratios (CCR) by multiplying the overall volume of communications by a constant factor. We use the CCR as the basis for our comparisons. There are many possible ways to define this ratio. We chose to define an average computation time t_{comp} by dividing the sum of all computation volumes by the average computational power of the platform. We similarly defined an average communication time t_{com} by dividing the sum of all communication volumes by the average bandwidth in the platform. We then define the CCR as the ratio $t_{\text{com}}/t_{\text{comp}}$.

Finally, we impose the first and last tasks of each task graph to be processed on the first processor P_{source}. P_{source} is then the processor used to communicate with the outside world: it receives the input data sets and sends back the results. These first and last tasks have a size 0 and correspond to the storage of input and output data. Our application settings includes both related and unrelated applications, as discussed in Section 4.2.1, but we do not distinguish them as they lead to comparable results.

4.5.3 Results

Study on small task graphs

In this set of experiments, we include all heuristics and the MLP algorithm which computes the optimal allocation. As different scenarios may lead to very different throughputs, we normalize all results so that the optimal single-allocation algorithm MLP has throughput one. An interesting question we want to answer is whether restricting to a single allocation limits the achievable throughput, compared to a strategy like HEFT. Top of Figure 4.3 shows that the optimal single-allocation strategy MLP is better than HEFT as soon as communications are not negligible, that is when the CCR is greater than 0.02, and better than Data-Parallel when the CCR is greater than 0.04. When the communication-to-computation ratio exceeds 0.05, HEFT and Data-Parallel do not exceed 70% of the optimal throughput of a single allocation. This both justifies our claim that static scheduling techniques can outperform classical schedulers, and motivates the search for single-allocation schedules. When the communication-to-computation ratio is very small (smaller than 0.01), communications are negligible and the best solution may be to execute a different instance of the task graph on each processor (except for peculiar applications whose tasks have strong unrelated characteristics). This explains the performance of HEFT for very low values of the CCR: it is able to use more resources. On the other hand, when the CCR is very high (larger than one), sometimes communications are so expensive that all tasks must be mapped on the source processor P_{source}. All these heuristics (Data-Parallel, Delegate and HEFT) that are able to detect this then deliver the optimal throughput.

In the second diagram of Figure 4.3 and in the first one of Figure 4.4, we compare the optimal solution using a single allocation (MLP) and the Delegate strategy to an upper bound of the throughput reachable using many allocations. On average, this upper bound is 40% better than the optimal mono-allocation solution, and this difference quickly tends to zero when the CCR increases. Thus, using a single allocation is often sufficient to reach a good throughput.

In the other diagrams of Figures 4.3 and 4.4, we compare the optimal allocation strategy MLP to the allocation-building heuristics described in Section 4.4. The Delegate heuristic always achieves the best throughput among the steady-state heuristics, except few cases dominated by Neighborhood. Furthermore, its performance is very close to the optimal performance defined by the mixed linear program (MLP). We also notice that strategies based on the rounding-

off of relaxed linear programs are not significantly more efficient than our greedy strategies, despite their higher complexity. NEIGHBORHOOD sometimes returns very high throughputs, but on average DELEGATE returns by far higher throughputs.

The attentive reader will notice that, sometimes, the DELEGATE heuristic builds an allocation with a higher throughput than the optimal allocation given by MLP. This apparent paradox does not contradict the optimality of MLP. Indeed, MLP gives an allocation with optimal *theoretical* throughput, and we have seen that the experimental throughput may slightly differ from the theoretical throughput. This is why DELEGATE appears sometimes to be "better than the optimal".

Study on larger task graphs

Figure 4.5 shows a comparison between some steady-state heuristics (REFINED_GREEDY and DELEGATE) and the HEFT strategy on larger task graphs. Due to the high running times of the RLP_MAX and RLP_RAND heuristics, we only ran them on a subset of the larger task graphs on which they happened to perform poorly. Therefore, we do not report here on their performance.

As we cannot compute the optimal single-allocation throughput anymore, we normalize all results so that DELEGATE gives a throughput of one. Note that the normalized throughput is exactly the inverse of the normalized *makespan*, as we defined the practical throughput as the ratio between the number of task graph instances and the makespan.

Greedy heuristics (SIMPLE_GREEDY and REFINED_GREEDY) and CLUSTERING perform similarly. Similarly to the small task graphs case, as soon as communications matter, DELEGATE provides the best results, while HEFT performs similarly to greedy strategies. However, when communications are almost negligible, greedy strategies and HEFT outperform DELEGATE. In other words, DELEGATE is the best heuristic in the complex cases, that is when communications do not impose trivial solutions (no parallelism when communications are too large, one task graph per processor when there are no communications). On average, DELEGATE achieves makespans 2.35 times shorter than those of HEFT, and 1.76 shorter than those of DATA-PARALLEL. On average, the upper bound is 15% larger than the throughput obtained by DELEGATE, showing that using a single allocation is acceptable to reach good throughputs. NEIGHBORHOOD offers worse results than DELEGATE, being on average 75% slower than DELEGATE. However, its results are mainly unpredictable, since they can be very good, being even better than DELEGATE, or very bad.

Running times

We also study the running times of the different heuristics. The CPLEX software [38] allows us to solve mixed linear programs in about one minute for the small settings. However, for task graphs larger than about 20 tasks, the running time is often more than a full day. Both RLP_MAX and RLP_RAND needs to solve many linear programs (over the rationals). Since these linear programs are quite big, the total running time of these heuristics often reach 10 minutes for the large settings. The greedy heuristics SIMPLE_GREEDY and REFINED_GREEDY are very fast (less than one second), whereas DELEGATE computes an allocation in about one half of the time needed for HEFT to compute its schedule on 1000 instances.

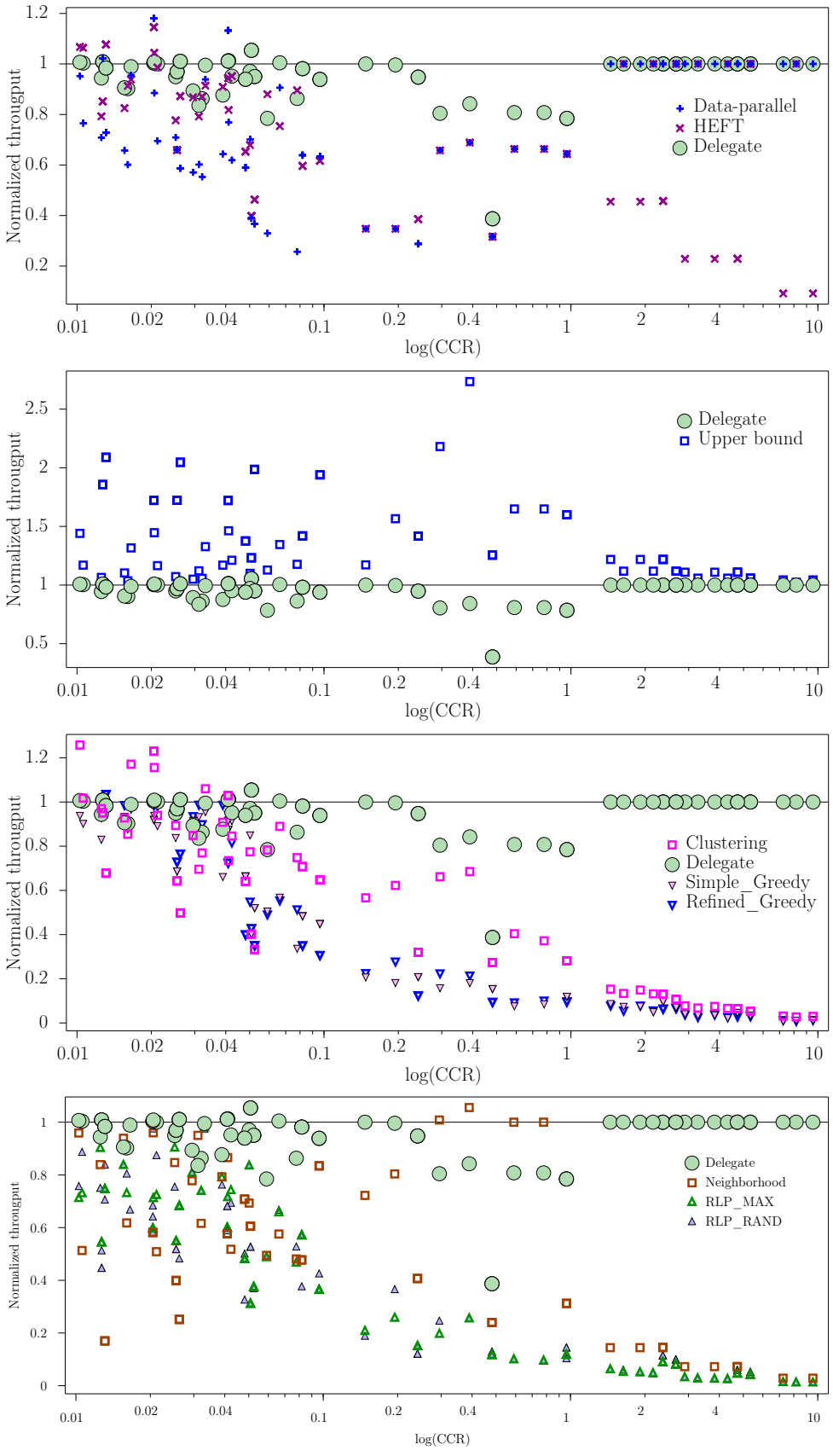


Figure 4.3: Performance on the small task graphs. Results are normalized such that MLP has throughput one.

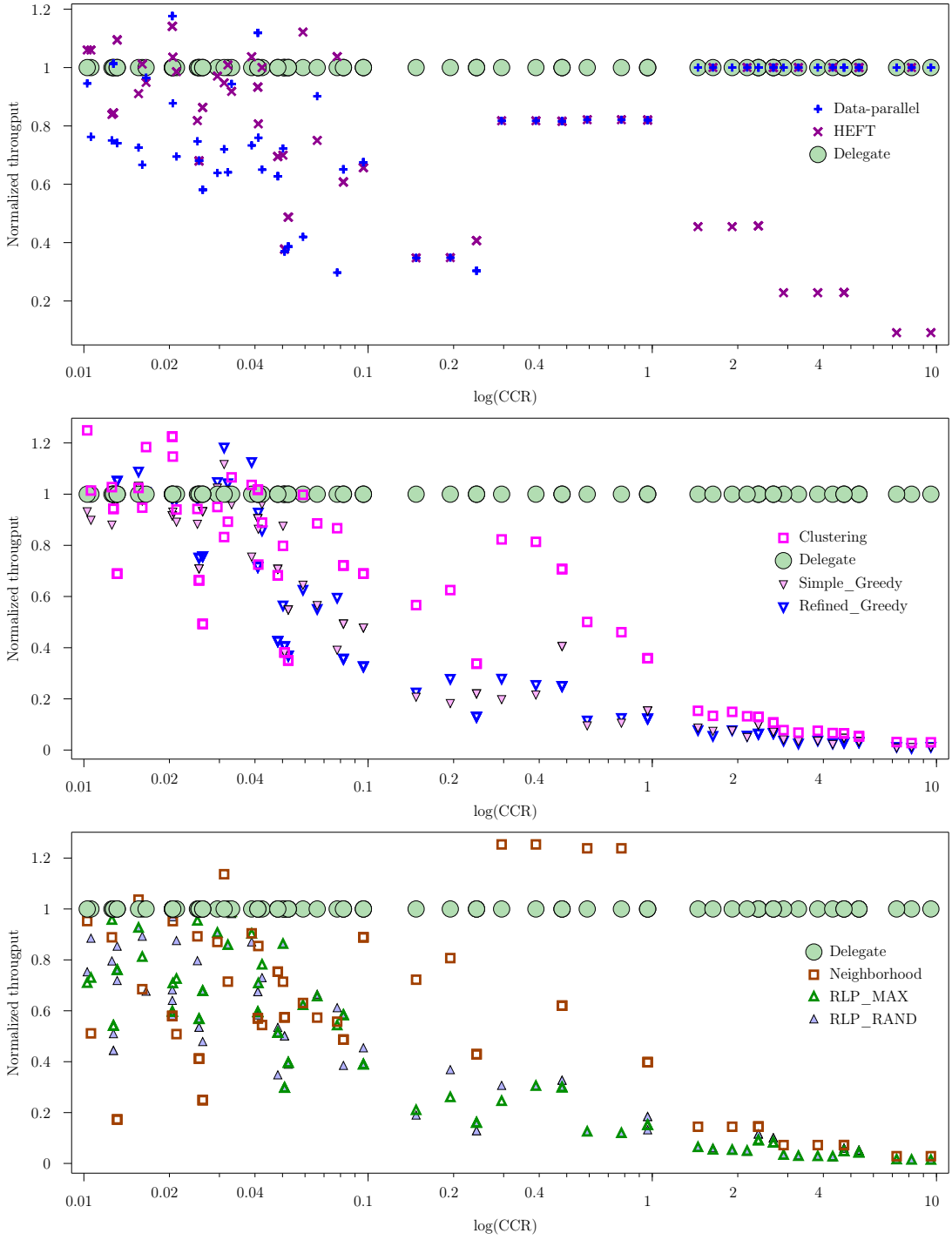


Figure 4.4: Performance on the small task graphs. Results are normalized such that DELEGATE has throughput one.

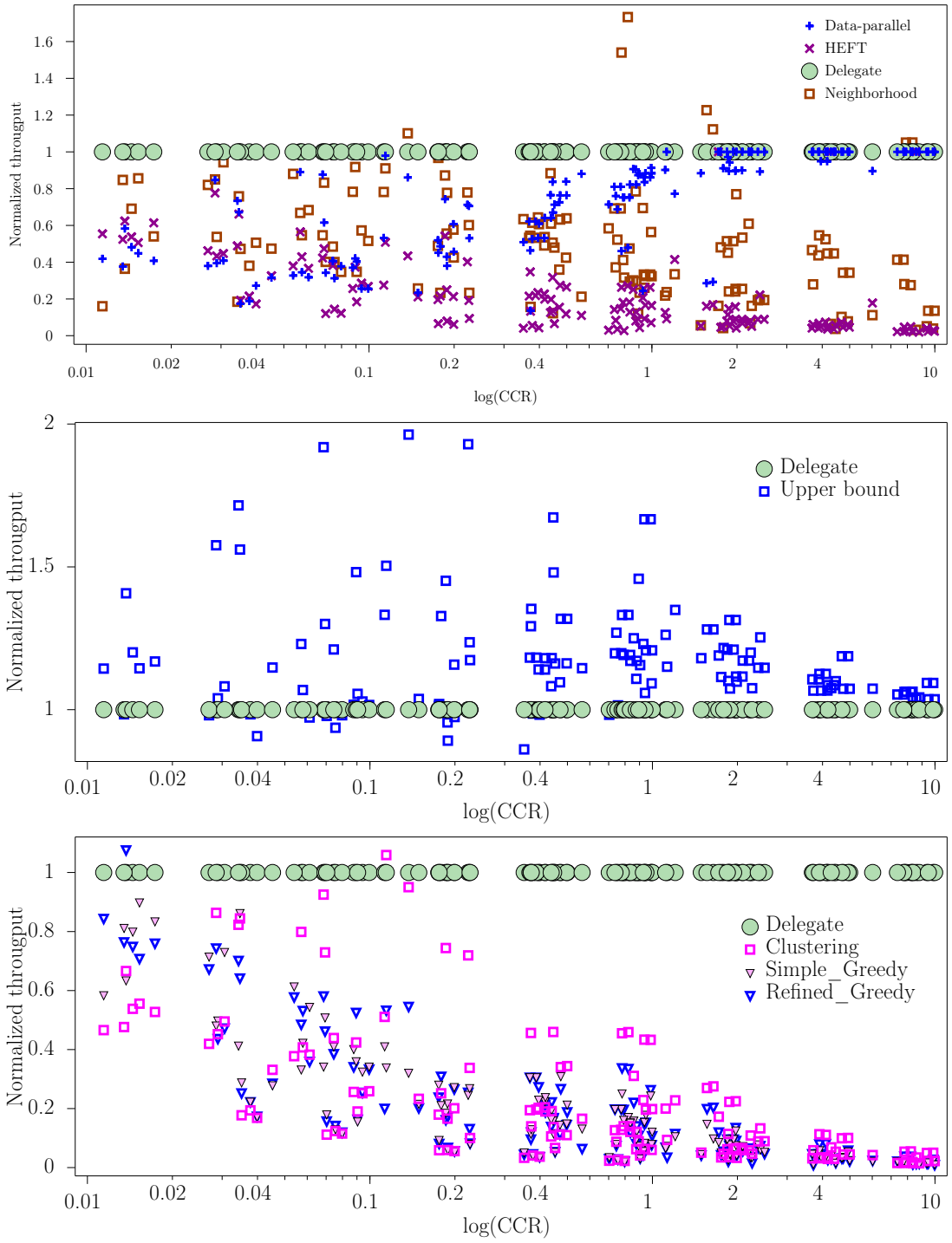


Figure 4.5: Performance on the larger task graphs. Results are normalized such that DELEGATE has throughput one.

4.6 Conclusion and perspectives

In this chapter, we have studied the scheduling of a collection of task graphs on a heterogeneous platform. Rather than attempting the classical makespan minimization, we have taken advantage of the regularity of our problem to optimize the system throughput by applying steady-state techniques. We have presented a mixed linear programming approach to compute the optimal allocation, and several heuristic algorithms to build practical solutions. We have performed extensive simulations to compare the performance of our heuristics to a classical scheduler, HEFT. Simulation results show the benefit of our approach as soon as communication times are not negligible. Our heuristic of choice, DELEGATE, almost always gives the best makespan. On average, DELEGATE achieves makespans which are twice shorter than HEFT's ones, in our simulation settings, while having a lower running time.

Chapter 5

Steady-state scheduling of dynamic bag-of-tasks applications

5.1 Introduction

In this chapter, we study a rather classical parallel scheduling problem: a set of independent tasks and a simple master–worker platform, i.e., a main processor, which initially owns all the tasks and redistributes them to a pool of secondary processors, or workers, which process the tasks. We aim at the average number of tasks processed by the platform per time unit. This kind of application, commonly called a bag of tasks, is an accurate model for numerous large-scale applications. It typically applies to problems processed through the world-wide BOINC framework [7] inspired by SETI@home [60, 8], or the similar Folding@home [61] project.

This simple but successful model has been studied for a long time and is now well-known. However, considering that all tasks have identical characteristics is a strong assumption, which is violated in many cases; for example, this is the case of projects deployed through the BOINC framework. Furthermore, most frameworks can simultaneously process multiple applications that are very different in nature: files to be processed, images to be analyzed, matrices to be inverted or multiplied, etc. Thus, a better model consists in using several types of tasks or several applications, given to the master and redistributed to the workers. Even if two tasks of the same type can have different characteristics, they should remain rather similar, especially their *communication-to-computation ratio* (CCR), while two tasks of different types may have completely different characteristics: a number to be factorized [37] requires a very small amount of data but a lot of computation, while an image to be analyzed may have a high CCR.

This model leads to a new challenge, since the scheduler needs to choose which application is processed by which worker. For example, tasks with a small amount of data but requiring a lot of computations should be delivered to workers with a poor bandwidth—like clients connected to the internet through a RTC connexion—while tasks with a large amount of data but a smaller amount of computation are well-suited to many computers linked by a high-speed enterprise network.

Currently, many bag-of-tasks applications use simple policies like an On-Demand dynamic algorithm or a Round-Robin distribution to distribute jobs to workers. While theses strategies offer good performance in many situations and computational power become cheaper and cheaper, one could wonder whether clever schedulers are still needed. In [22], Benoit et al. expose situations with several constant bag-of-task applications requiring such schedulers. In this chapter, we study the impact of small variations between the multiple instances of applications

to schedule, to assess the limits of a static schedule in this context.

In Section 5.2, platform and application models are described. We first give notations for constant applications, i.e., all instances of a given application are identical, even if several different applications need to be scheduled, and then we present our stochastic formulation used to model the variations between instances. Similarly, Section 5.3 first solves the simpler, constant case, before describing an ε-approximation and several heuristics, which all apply to the stochastic formulation. In Section 5.4, both heuristic and approximation are compared to the On-Demand and Round-Robin algorithms, showing that a little knowledge about applications is sufficient to really improve scheduling results. We conclude in Section 5.5 with some final remarks.

5.2 Notations and models

In this section, we refine the model presented in Section 3 and we expose some notations specific to this chapter.

5.2.1 Platform model

We focus on star-shaped platforms, made of a master computer P_0 linked to $n-1$ worker processors $P_1, \ldots, P_{n-1}$. Since bag-of-tasks applications are often deployed over the internet or on a computing grid, computing resources are mainly heterogeneous. This heterogeneity is modeled by the speed s_i and the bandwidth bw_i of processor P_i.

Currently, multi-core technology is used by almost all processors, especially those in file servers, while modern network cards with onboard processors enable network links to be shared by several incoming and outgoing communications. The bounded multi-port communication model presented by Hong and Prasanna [54] states that the number of simultaneous communications is unbounded, the sum of the bandwidths of these communications being limited by the bandwidth of the network card. In our case, only the master P_0 can communicate to several other processors, as long as its bandwidth bw_0 is not exceeded. Recent versions of communication libraries such as MPICH2 [59] are now multithreaded and allow such concurrent communications.

5.2.2 Constant applications model

As we said in Section 5.1, we gather all tasks into several applications. First, we assume that all instances of a given application have identical characteristics. In this chapter, contrarily to the previous ones, we do not need a graph representation anymore since all tasks are independent. Thus, T_k denotes the set of tasks of application k (with $1 \leq k \leq m$), and all instances are defined by a volume of communication $V_{comm}(k)$ in bytes and a volume of computation $V_{comp}(k)$ in Flops. An instance of application T_k is sent in time $V_{comm}(k)/b$, where b is the average bandwidth allocated to the communication, and computed by worker P_i in time $V_{comp}(k)/s_i$. We allow overlap of computations by communications: any worker can work at the already received data, while it is retrieving the next instances from the master. This hypothesis is consistent with the bounded multi-port communication model, as it is enabled by the same multi-core machines and multithreaded technologies. The master does not participate in the work, acting as a dedicated file server.

Following the steady-state approach, each application is assumed to have a really large number of instances, allowing us to consider a continuous flow rather than a finite number of

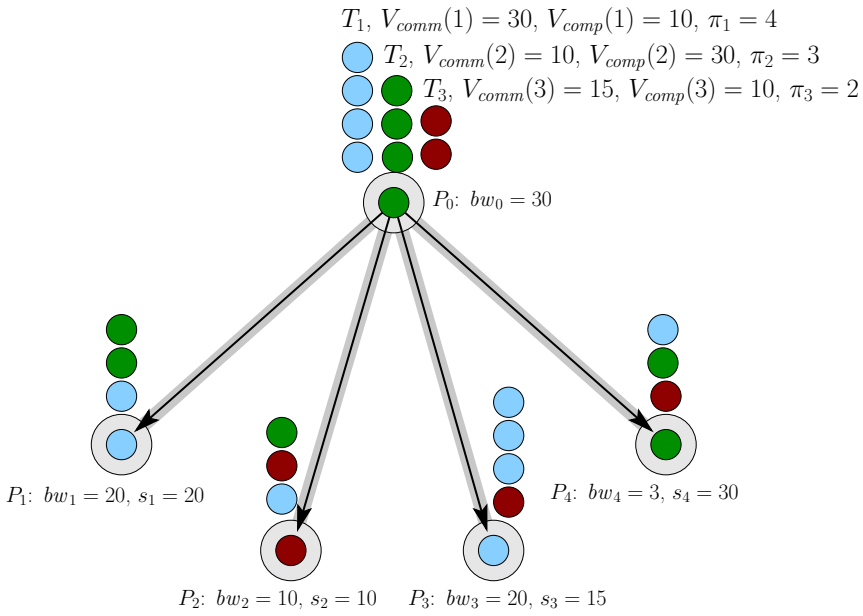


Figure 5.1: Example A: platform graph, made of a master P_0 and 4 workers, and 3 applications.

instances. The overall number of instances is *a priori* different from an application to another. To cope with this new parameter, each application has a positive priority π_k : if application T_1 has a priority $\pi_1 = 1$ and T_2 has another priority $\pi_2 = 2$, then the throughput of application T_2 must be twice the throughput of application T_1. In the example shown in Figure 5.1, we have three applications, with respective priorities equal to 4, 3 and 2.

We still aim at maximizing the throughput of our platform. Due to the priorities of tasks, we enforce the constraint that the throughput ρ^k of an application k is proportional to its priority: if application T_1 has a priority π_1 and T_2 has a priority π_2, then we have $\rho^1/\pi_1 = \rho^2/\pi_2$. Any processor P_i participates in the global throughput ρ^k of T_k, and we denote its participation by ρ_i^k.

Finally, we can distinguish two models, depending on the knowledge we have about our application:

***off-line* model:** in this model, we assume to know the future: all instances, as well as their volumes of computation or their data sizes, are known before the execution of the scheduling policy,

***online* model:** in this other model, instances arrive in the system while the algorithm is being executed, and the scheduler needs to map an incoming instance on the fly.

5.2.3 A stochastic formulation for dynamic applications

In the previous section, we presented a model using several applications with identical instances. Although this model is representative of many real cases, some applications deal with instances with different characteristics: the time required to factorize a number greatly varies from a number to another one, even if they are close. A first solution to cope with this problem is to split the global set of tasks into different "virtual" applications of identical data size and

computation time, even if computations have different natures. However, this can lead to almost as many virtual applications as instances, and it can be hard to predict the exact characteristics of an incoming task. A common way to deal with such an uncertainty is to model the size and the volume of computation of our tasks by random variables. Obviously, we could model instances of all applications by a single random variable for communication sizes, and another one for computation times. However, it seems reasonable to keep two varying characteristics for each application, one for computations and one for communications, especially if different applications have very different characteristics. In the following, we still have m different applications, and T_k (with $1 \leq k \leq m$) denotes such an application.

Thus, we define the two following notations:

- X^k_{comm} is a random variable, such that $X^k_{comm}(u)$ is the data size of the u-th instance of the k-th application.
- X^k_{comp} is another random variable, such that $X^k_{comp}(u)$ is the volume of computation of the u-th instance of the k-th application.

We assume we know a lower bound and upper bound for both communication and computation sizes of any application. Lower bounds are respectively denoted by $\min^k_{comm}$ and $\min^k_{comp}$, while upper bounds are denoted by $\max^k_{comm}$ and $\max^k_{comp}$. Finally, we need to know the following probability distributions:

- $\forall k, \forall c_1 \geq 0, \forall c_2 \geq c_1, \mathcal{P}\left(c_1 \leq X^k_{comp} \leq c_2\right)$,
- $\forall k, \forall d_1 \geq 0, \forall d_2 \geq d_1, \mathcal{P}\left(d_1 \leq X^k_{comm} \leq d_2\right)$,
- $\forall k, \forall c_1, d_1 \geq 0, \forall c_2 \geq c_1, \forall d_2 \geq d_1, \mathcal{P}\left(c_1 \leq X^k_{comp} \leq c_2; d_1 \leq X^k_{comm} \leq d_2\right)$.

There is no reason for X^k_{comm} and X^k_{comp} to be independent for any application T_k ; on the contrary, one can imagine that a small file would require less computation than a larger task, or can be synonymous of more computations than a larger one, for example due to a higher compression of the data.

5.3 Approximation and heuristics

In this section, we explain the solution given in [22], before giving an ε-approximation and heuristics adapted to the dynamic formulation.

5.3.1 Resolution of the constant case

First, we recall the solution to the problem, assuming that all instances of a given application are identical. We can easily write all the constraints that apply when we look at the throughput of our platform, i.e., the average number of instances of any application processed in one time unit. As we said in Section 5.2.2, the total throughput of application T_k is the sum of the contributions of all processors to this application:

$$\forall T_k, \sum_{1 \leq i < n} \rho^k_i = \rho^k.$$

We know that the throughput of T_k is proportional to its priority, and this can be written as:

$$\forall T_k, \frac{\rho^k}{\pi_k} = \frac{\rho^1}{\pi_1}.$$

The sum of all computation times on any worker per time-unit takes less than one time unit:

$$\forall 1 \leq i < n, \sum_{T_k} \rho_i^k \frac{V_{comp}(k)}{s_i} \leq 1.$$

Similarly, the sum of all incoming communication times on any worker per time-unit takes less than one time unit:

$$\forall 1 \leq i < n, \sum_{T_k} \rho_i^k \frac{V_{comm}(k)}{bw_i} \leq 1.$$

Finally, the master cannot exceed its own bandwidth:

$$\sum_{1 \leq i < n} \sum_{T_k} \rho_i^k \frac{V_{comm}(k)}{bw_0} \leq 1.$$

All these constraints form a rational linear program given in (5.1).

$$\begin{cases} \text{MAXIMIZE } \rho^1 \text{ UNDER THE CONSTRAINTS} \\ \text{(5.1a)} \quad \forall T_k, \quad \sum_{1 \leq i < n} \rho_i^k = \rho^k \\ \text{(5.1b)} \quad \forall T_k, \quad \frac{\rho^k}{\pi_k} = \frac{\rho^1}{\pi_1} \\ \text{(5.1c)} \quad \forall 1 \leq i < n, \quad \sum_{T_k} \rho_i^k \frac{V_{comp}(k)}{s_i} \leq 1 \\ \text{(5.1d)} \quad \forall 1 \leq i < n, \quad \sum_{T_k} \rho_i^k \frac{V_{comm}(k)}{bw_i} \leq 1 \\ \text{(5.1e)} \quad \sum_{1 \leq i < n} \sum_{T_k} \rho_i^k \frac{V_{comm}(k)}{bw_0} \leq 1 \end{cases} \tag{5.1}$$

All variables are rational, so we can find the solution in polynomial time [58]. Figure 5.2 presents a solution of this linear program applied to example A (itself presented on Figure 5.1). As we can see, P_3 and P_4 are kept idle, due to their low bandwidth, while communication links of P_1 and P_2 are saturated.

5.3.2 An ε-approximation

We previously presented a solution to our problem when all instances of a given application are identical. When we have variations between instances, one could argue that it should be sufficient to split each application into as many different, *virtual*, applications as there are different types of instances. Nonetheless, this solution would mainly lead to two new issues:

- the number of variables will dramatically increase, because we may end up with as many virtual applications as tasks,
- even if we consider continuous flows of instances, actual applications have a finite number of instances, and a large number of virtual applications means small numbers of instances, in contradiction with the steady-state hypothesis.

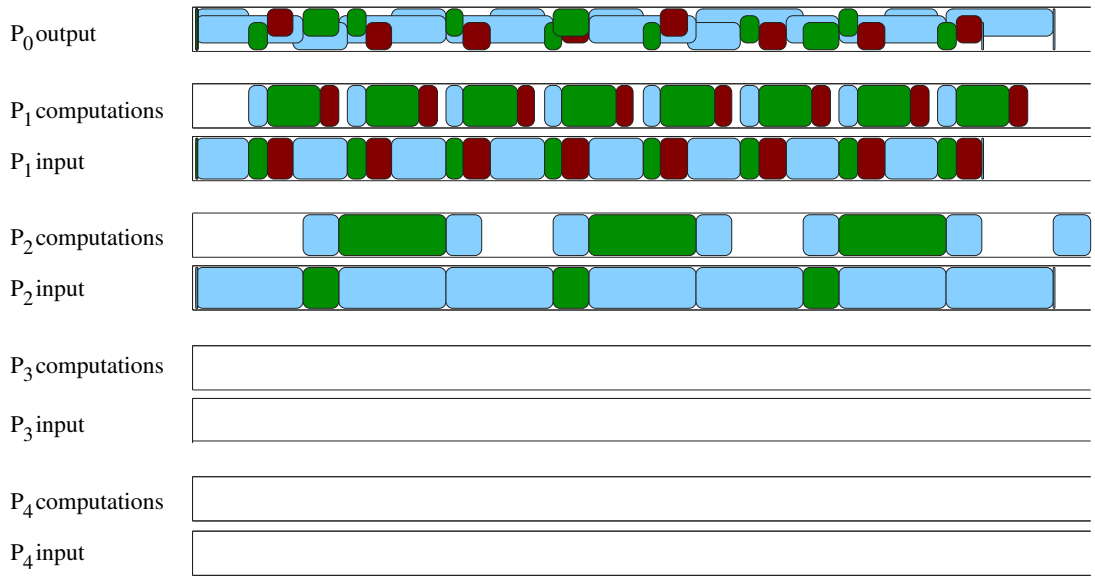


Figure 5.2: Execution of the first instances of Example A.

However, we recall that we assumed to know the distribution of computation and communication sizes of the instances of any application, as well as lower and upper bounds, as stated in Section 5.2.3. This knowledge is sufficient to propose an ε-approximation for our problem. In other words, if ε is any positive number, we are able to compute a solution whose throughput ρ is at least equal to $\rho^*/(1+\varepsilon)$, where ρ^* is the throughput of any optimal solution.

The idea underlying this ε-approximation is to split each application into several virtual applications such that two instances of the same virtual applications only have small differences in term of data sizes and computation volumes.

Before going further, we need to define several notations. Let γ_q^k be equal to $(1+\varepsilon)^q \min_{comp}^k$, with q between 0 and $Q^k = 1 + \left\lfloor \frac{\ln\left(\frac{\max_{comp}^k}{\min_{comp}^k}\right)}{\ln(1+\varepsilon)} \right\rfloor$, and δ_r^k be equal to $(1+\varepsilon)^r \min_{comm}^k$, with r between 0 and $R^k = 1 + \left\lfloor \frac{\ln\left(\frac{\max_{comm}^k}{\min_{comm}^k}\right)}{\ln(1+\varepsilon)} \right\rfloor$. An instance of application T_k is said to be in the interval $I_{q,r}^k = \left[\gamma_q^k; \gamma_{q+1}^k\right] \times \left[\delta_r^k; \delta_{r+1}^k\right]$ if its volume of computation is comprised between γ_q^k and γ_{q+1}^k and its data size is between δ_r^k and δ_{r+1}^k. Finally, let $\rho_{i,q,r}^k$ be the contribution of processor P_i to the throughput of instances of application T_k in the interval $I_{q,r}^k$. On Figure 5.3, an example with 3 applications, similar to Example A, shows the partition of a small set of instances with $\varepsilon = 0.4$. As we can see, some intervals remain empty.

To simplify equations, we denote by $p_{q,r}^k$ the probability of an instance of application T_k to be in a given interval $I_{q,r}^k$:

$$p_{q,r}^k = \mathcal{P}\left(\gamma_q^k \leq X_{comp}^k < \gamma_{q+1}^k; \delta_r^k \leq X_{comm}^k < \delta_{r+1}^k\right).$$

By construction, we have:

$$\forall k, \sum_{q,r} p_{q,r}^k = 1. \tag{5.2}$$

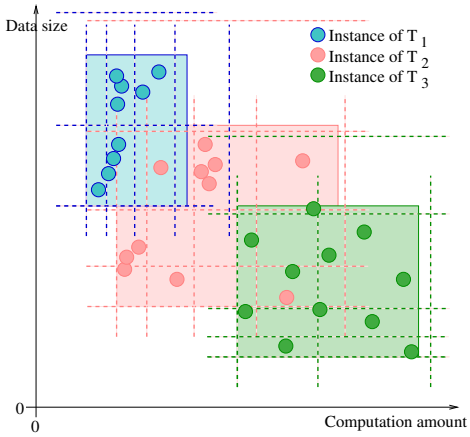


Figure 5.3: Example A bis: partition of instances with $\varepsilon = 0.4$.

Our hypotheses ensure that if there are t instances of application T_k, then we expect $t \cdot p_{q,r}^k$ instances in the interval $I_{q,r}^k$. These instances constitute the virtual application $T_{k,q,r}$.

The total throughput of any virtual application of T_k is the sum of the contributions of all processors to this virtual application. The total throughput of any virtual application $T_{k,q,r}$ of T_k is also equal to the throughput of the whole application T_k times the probability of an instance to be in the interval $I_{q,r}^k$:

$$\forall k, \forall q < Q^k, \forall r < R^k, \sum_{1 \leq i < n} \rho_{i,q,r}^k = p_{q,r}^k \rho^k.$$

The throughput of T_k is still proportional to its priority:

$$\forall T_k, \frac{\rho^k}{\pi_k} = \frac{\rho^1}{\pi_1}.$$

Since the precise computation and communication sizes are not available, we overestimate them, ensuring the feasibility of our schedule. By definition, the computation of any instance of application T_k in the interval $I_{q,r}^k$ takes less than $\frac{\gamma_{q+1}^k}{s_i}$ on processor P_i. The sum of all computations, on any worker and per time-unit, must take less than one time unit. We enforce a stronger constraint by considering that all instances in the interval $I_{q,r}^k$ take a time exactly equal to $\frac{\gamma_{q+1}^k}{s_i}$:

$$\forall 1 \leq i < n, \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho_{i,q,r}^k \frac{\gamma_{r+1}^k}{s_i} \right) \leq 1.$$

Similarly, the communication to P_i of any instance of application T_k in the interval $I_{q,r}^k$ takes less than $\frac{\delta_{q+1}^k}{bw_i}$, and the sum of all incoming communications of any worker takes less than one time unit. Again, we use a stronger constraint, by considering that all communications take exactly $\frac{\delta_{q+1}^k}{bw_i}$ time units:

$$\forall 1 \leq i < n, \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho_{i,q,r}^k \frac{\delta_{r+1}^k}{bw_i} \right) \leq 1.$$

Finally, the master cannot exceed its own bandwidth:

$$\sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho^k_{i,q,r} \frac{\delta^k_{r+1}}{bw_0} \right) \leq 1.$$

All theses constraints can be gathered into the rational linear program (5.3):

$$\begin{cases} \text{MAXIMIZE } \rho = \rho^1 \text{ UNDER THE CONSTRAINTS} \\ (5.3a) \quad \forall T_k, \forall q < Q^k, \forall r < R^k, \quad \sum_{1 \leq i < n} \rho^k_{i,q,r} = p^k_{q,r} \rho^k \\ (5.3b) \quad \forall T_k, \quad \frac{\rho^k}{\pi_k} = \frac{\rho^1}{\pi_1} \\ (5.3c) \quad \forall 1 \leq i < n, \quad \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho^k_{i,q,r} \frac{\gamma^k_{q+1}}{s_i} \right) \leq 1 \\ (5.3d) \quad \forall 1 \leq i < n, \quad \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho^k_{i,q,r} \frac{\delta^k_{r+1}}{bw_i} \right) \leq 1 \\ (5.3e) \quad \sum_{1 \leq i < n} \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho^k_{i,q,r} \frac{\delta^k_{r+1}}{bw_0} \right) \leq 1 \end{cases} \tag{5.3}$$

Theorem 5.1. *An optimal solution of Linear Program (5.3) describes a solution with a throughput ρ larger than $\rho^*/(1+\varepsilon)$, where ρ^* is the optimal throughput of application T_1.*

Proof. The proof contains three steps: we first show that this linear program returns a valid solution, we then compute an upper bound $\rho_{\max}$ of the optimal throughput ρ^*, and finally, we show that the throughput ρ of the solution is larger than $\rho^*/(1+\varepsilon)$.

1. Consider the N^1 first instances of application T_1. Due to the priorities, we also consider the $N^k = \left\lfloor \frac{\pi_k}{\pi_1} N^1 \right\rfloor$ first instances of application T_k (with $1 \leq i \leq m$). By definition of the $p^k_{q,r}$'s, if $n^k_{q,r}$ denotes the actual number of instances in the interval $I^k_{q,r}$ among the first N^k instances of T_k, we have:

$$p^k_{q,r} = \lim_{N^1 \to \infty} \frac{n^k_{q,r}}{N^k}. \tag{5.4}$$

   Linear program (5.3) is equivalent to linear program (5.1) with $\sum_k \left(Q^k R^k\right)$ applications named $T_{k,q,r}$ and defined by:

   - $V_{comm}(k)_{q,r} = \delta^k_{q,r}$,
   - $V_{comp}(k)_{q,r} = \gamma^k_{q,r}$,
   - $\pi_{k,q,r} = p^k_{q,r} \pi_k$.

   Let (q_0, r_0) be such that $p^1_{q_0,r_0}$ is positive. We know [22] that linear program (5.1) returns a valid schedule σ for this set of new applications, ensuring a throughput $\rho^k_{q,r}$ to the application $T_{k,q,r}$:

$$\rho^k_{q,r} = \frac{\pi_k p^k_{q,r}}{\pi_1 p^1_{q_0,r_0}} \rho^1_{q_0,r_0}.$$

If we use σ to process $\left\lfloor p^1_{q_0,r_0} N^1 \right\rfloor$ instances of T_{1,q_0,r_0} in time $\mathcal{T}_{N^1}$, then $\left\lfloor p^k_{q,r} N^k \right\rfloor$ instances of $T_{k,q,r}$ are concurrently processed, and we have:

$$\rho^k_{q,r} = \lim_{N^1 \to \infty} \frac{\left\lfloor p^k_{q,r} N^k \right\rfloor}{\mathcal{T}_{N^1}} = \lim_{N^1 \to \infty} \frac{p^k_{q,r} N^k}{\mathcal{T}_{N^1}}.$$

Since an instance in the interval $I^k_{q,r}$ has by definition a size less than $(\delta^k_{q+1,r+1}, \gamma^k_{q+1,r+1})$, σ can process the $\left\lfloor p^k_{q,r} N^k \right\rfloor$ first instances of the original application T_k found in each interval $I^k_{q,r}$ in time $\mathcal{T}_{N^1}$. Thanks to Equation (5.4), we have:

$$\rho^k_{q,r} = \lim_{N^1 \to \infty} \frac{n^k_{q,r}}{\mathcal{T}_{N^1}}.$$

Thus, for each virtual application, an optimal solution of (5.3) describes the average number of instances processed by the resources, allowing to reach the desired throughput equal to $\rho^k = \sum_{q,r} \rho^k_{q,r}$ to application T_k. By definition of ρ^*, we have $\rho \leq \rho^*$.

2. An upper bound $\rho_{\max}$ of the optimal throughput is easily obtained, by solving the linear program (5.5). This linear program corresponds to the problem obtained by replacing any instance of application T_k belonging to interval $I^k_{q,r}$ by a smaller instance of size $(\delta^k_{q,r}, \gamma^k_{q,r})$. Thus, the optimal throughput for this new set of applications is larger than the one of the original applications. Since this linear program returns the optimal throughput of this new set, we have an upper bound of the optimal throughput, which can be reached with our original applications T_k's:

$$\rho^* \leq \rho_{\max}.$$

$$\begin{cases} \text{MAXIMIZE } \rho_{\max} = \rho'^1 \text{ UNDER THE CONSTRAINTS} \\ \text{(5.5a)} \quad \forall T_k, \forall q < Q^k, \forall r < R^k, \quad \sum_{1 \leq i < n} \rho'^k_{i,q,r} = p^k_{q,r} \rho'^k \\ \text{(5.5b)} \quad \forall T_k, \quad \frac{\rho'^k}{\pi_k} = \frac{\rho'^1}{\pi_1} \\ \text{(5.5c)} \quad \forall 1 \leq i < n, \quad \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho'^k_{i,q,r} \frac{\gamma^k_q}{s_i} \right) \leq 1 \\ \text{(5.5d)} \quad \forall 1 \leq i < n, \quad \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho'^k_{i,q,r} \frac{\delta^k_r}{bw_i} \right) \leq 1 \\ \text{(5.5e)} \quad \sum_{1 \leq i < n} \sum_{T_k} \sum_{q<Q^k, r<R^k} \left(\rho'^k_{i,q,r} \frac{\delta^k_r}{bw_0} \right) \leq 1 \end{cases} \tag{5.5}$$

3. By definition, we have $\delta^k_{r+1} = (1+\varepsilon)\delta^k_r$ and $\gamma^k_{q+1} = (1+\varepsilon)\gamma^k_q$. Since all constraints are linear, one can easily check that $\left(\rho'^k_{i,q,r}\right)_{k,i,q,r}$ is a solution of (5.5) if, and only if, $\left(\rho^k_{i,q,r} = \rho'^k_{i,q,r}/(1+\varepsilon)\right)_{k,i,q,r}$ is a solution of (5.3). Using Equations (5.2) and (5.3a), we have $\rho^1 = \sum_{i,q,r} \rho^1_{i,q,r}$, leading to $\rho'^1 = \rho^* = (1+\varepsilon)\rho$.

Altogether, we have $\rho \leq \rho^* \leq (1+\varepsilon)\rho$: the throughput ρ is larger than $\rho^*/(1+\varepsilon)$, concluding the demonstration.

■

5.3.3 Heuristic solutions to the online problem

In the previous subsection, we presented an ε-approximation requiring a lot of knowledge about the application to schedule, as said in Section 5.2.3. If determining a lower bound and an upper bound on both communication and computation sizes is often realistic, determining the distribution of the communication and computation sizes involves a lot of informations on the tasks, which can be unreachable to the user. Moreover, when a new instance arrives in the system, even if its data size could be easily determined, its volume of computation could remain undetermined. Thus, we can distinguish several cases depending on the available knowledge:

- All communication and computation sizes are known before the execution of the algorithm. This is the *off-line*, or the *clairvoyant* case.
- The *semi-clairvoyant* case: both communication and computation sizes of an instance are known when it is submitted to the system; the computation volume may be deduced more or less precisely from the communication size,
- The computation volume is not predictable when the instance is submitted, only the communication size is known; this is the *non-clairvoyant* case.

To circumvent these difficulties, we propose several *online* heuristics, based on the previous ε-approximation. All of them use the first instances as a sample to get a sufficient knowledge about the distributions of the random variables.

These heuristics use the first instances to obtain a preview of the involved distributions, and then to split the following instances in several buckets following different methods. In the following lines, we consider an example with a sample set of eight values $\{\min_{\text{pre}} = 1, 2, 4, 5, 6, 8, 9, 13 = \max_{\text{pre}}\}$ to split into $s = 4$ buckets:

arithmetical: buckets are defined such that the difference of size between two instances in the same bucket is less than $\frac{\max_{\text{pre}} - \min_{\text{pre}}}{s}$. In our example, the first bucket contains values between 1 and 4, the second one values between 4 and 7, the third one values between 7 and 10, the last one values between 10 and 13. An example of this method is shown in Figure 5.4, with three applications split into 3×3 buckets.

geometrical: buckets are defined such that the ratio between two instances in the same bucket is less than $\sqrt[s]{\frac{\max_{\text{pre}}}{\min_{\text{pre}}}}$. The four buckets for our example are $[1; 1.89]$, $[1.89; 3.60]$, $[3.60; 6.84]$, and $[6.84; 13]$.

recursive: at each of the $\log(s)$ steps, we split our values into two buckets with the same number of values. This method requires s to be a power of two. In our case, buckets are $(1, 2)$, $(4, 5)$, $(6, 8)$, and $(9, 13)$. Only distinct values are taken into account to form these buckets. If the size of an incoming instance falls outside these buckets, we assume that this instance belongs to its nearest bucket. In our case, this is equivalent to consider that the buckets are $[0; (2+4)/2[$, $[(2+4)/2; (5+6)/2[$, $[(5+6)/2; (8+9)/2[$, and $[(8+9)/2; \infty[$.

The number of instances in each bucket divided by the size of the sample gives us an indication of the probability required to solve our linear program. In our experiments, we assumed that the computation volume could not be determined from the data size, thus we only used communication sizes for the split.

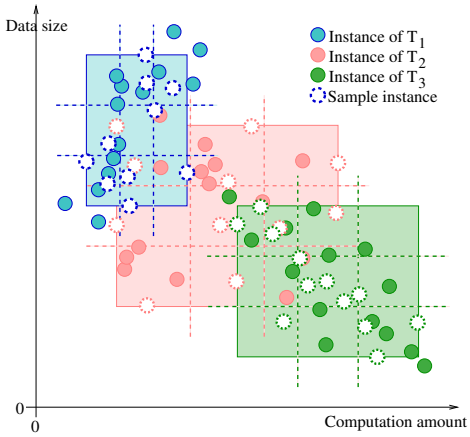


Figure 5.4: Partition of samples in 3×3 buckets.

5.3.4 Reference heuristics

To assess the quality of our heuristics, we compare them to two classical scheduling policies:

Round-Robin : incoming instances are distributed in turn to each worker,

On-Demand : each instance is allocated to the first available worker. To allow an overlap of computations by communications, workers maintain a buffer of b tasks: while an instance is processed, any worker can receive and store at most b instances.

Contrarily to our static methods, Round-Robin and On-Demand do not take care of the differences between workers, but the dynamicity of On-Demand allows to not overload slow workers. Moreover, as soon as only communications or computations matter, then On-Demand offers very good performance.

Theorem 5.2. *As soon as communications (or, symmetrically, computations) are dominant, the* On-Demand *heuristic returns asymptotically optimal schedules. More formally, communications are said dominant if the smallest communication time is larger than the largest computation time:*

$$\frac{\min_{1\leq k\leq m}\left(V_{comm}(k)\right)}{\max_{0\leq i\leq n}\left(bw_i\right)} \geq \frac{\max_{1\leq k\leq m}\left(V_{comp}(k)\right)}{\min_{1\leq i\leq n}\left(s_i\right)}.$$

Computations are said dominant if we have:

$$bw_0 \geq \sum_{i=1}^{n} bw_i \quad \text{and} \quad \frac{\min_{1\leq k\leq m}\left(V_{comp}(k)\right)}{\max_{1\leq i\leq n}\left(s_i\right)} \geq \frac{\max_{1\leq k\leq m}\left(V_{comm}(k)\right)}{\min_{0\leq i\leq n}\left(bw_i\right)}.$$

The constraint on the bandwidth of the master is required since we do not specify the communication policy in case of contentions, and a worker could suffer from starvation, leading to very large communication times.

Proof. In the following proof, we use the following notations:

- b is the size of the buffer used to store incoming instances on each processor,
- bw_{tot} is the sum of the bandwidths of all workers ($bw_{tot} = \sum_{i=1}^{n} bw_i$),

- N is the number of scheduled instances: $N^k = \pi_k N$ is the number of instances of T_k,
- $\rho^*(N)$ is the optimal throughput that can be obtained, and $T^*(N)$ is the associated makespan ($\rho^*(N) = T^*(N)/N$),
- $T(N)$ is the makespan achieved by an On-Demand schedule, and $\rho(N)$ is associated throughput ($\rho(N) = T(N)/N$),
- $\Gamma = \frac{\max_{1\leq k\leq m}\left(\max^k_{comp}\right)}{\min_{1\leq i\leq n}(s_i)}$ is the maximum computation time,
- $\Delta = \frac{\max_{1\leq k\leq m}\left(\max^k_{comm}\right)}{\min_{1\leq i\leq n}(bw_i)}$ is the maximum communication time,
- T_i is the completion time of processor P_i,
- t is the minimum completion time among all processors,
- Q_i (respectively, R_i) is the volume of communications (respectively, computations) given to P_i,
- Q (respectively, R) is the total volume of communications (respectively, computations).

By definition, we have: $Q = \sum_{k=1}^{m} \left(N\pi_k V_{comm}(k)\right) = \sum_{i=1}^{n} Q_i$ and $R = \sum_{k=1}^{m} \left(N\pi_k V_{comp}(k)\right) = \sum_{i=1}^{n} R_i$.

Dominating computations. First, we have $bw_0 \geq \sum_{i=1}^{n} bw_i$. Thus, the master can simultaneously communicate with all workers without being slowed down by its own bandwidth: each worker begins its work at the latest at time Δ. The reception of a single instance takes less time that the computation of any other instance (by definition of dominating computations): as soon as a worker begins its work, its communications are overlapped by computations and it continuously processes instances until its termination.

Let P_j be the first worker to terminate ($T_j = t$). Since P_j finishes at time t, there is no more instances to process at this time. Moreover, just before time t, any P_k may have solicited for at most b instances. So, P_k has at most $b+1$ instances to process, counting the instance which is handled at time t, and P_k finishes its work before time $t + (b+1)\Gamma$:

$$T(N) \leq t + (b+1)\Gamma \Leftrightarrow T(N) - (b+1)\Gamma \leq t. \tag{5.6}$$

The processing time of P_i is R_i/s_i. Since P_i begins its work before time Δ, we have:

$$\forall i \geq 1, t \leq T_i \leq \frac{R_i}{s_i} + \Delta \Rightarrow s_i\,(t - \Delta) \leq R_i.$$

Summing these inequalities and applying (5.6) gives:

$$\sum_{i=1}^{n} s_i(T(N) - (b+1)\Gamma - \Delta) \leq \sum_{i=1}^{n} s_i(t - \Delta) \leq \sum_{i=1}^{n} R_i = R. \tag{5.7}$$

A lower bound on the makespan associated to an optimal throughput is given by:

$$T^*(N) \geq \frac{R}{\sum_i s_i} = N\left(\frac{\sum_k \pi_k V_{comp}(k)}{\sum_{i=1}^{n} s_i}\right). \tag{5.8}$$

Using Equations (5.7) and (5.8) leads to:

$$\left(\sum_{i=1}^{n} s_i\right)(T(N) - (b+1)\Gamma - \Delta) \leq R \leq \left(\sum_{i=1}^{n} s_i\right) T^*(N).$$

And then:

$$\frac{T(N)}{T^*(N)} \leq 1 + \frac{(b+1)\Gamma + \Delta}{T^*(N)}.$$

Thanks again to Equation (5.8), we have $\lim_{N\to\infty} T^*(N) = \infty$ and then $\frac{T(N)}{T^*(N)} \leq 1 + o_{N\to\infty}(1)$. Thus, On-Demand is asymptotically optimal when computations dominate:

$$\lim_{N\to\infty} \frac{\rho(N)}{\rho^*(N)} = 1.$$

Dominating communications. The reception of a single instance takes more time that the computation of another instance (by definition of dominating communications): as soon as a worker begins to receive data at time 0, it is kept continuously receiving data until its last communication, immediately followed by the computation of the last instance. Only the computation of the last instance is not overlapped by communications.

Since the master bandwidth is shared among all workers, two cases needs to be studied.

1. The master bandwidth does not limit communications: $bw_0 \geq \sum_{i=1}^{n} bw_i$.
   This first case is quite similar to the previous situation. Assume that P_k is still the last processor to finish ($T_i = T(N)$), while P_j is the first one ($T_j = t$). Since P_j finishes at time t, there is no more instances to process at this time. Moreover, just before time t, P_k may have requested for at most b instances, and it finishes its work before time $t + b\Delta + \Gamma$:

   $$T(N) \leq t + b\Delta + \Gamma \Leftrightarrow T(N) - b\Delta - \Gamma \leq t. \tag{5.9}$$

   The total communication time of P_i is Q_i/bw_i. P_i begins its communications at time 0 and finishes its work before time $Q_i/bw_i + \Gamma$:

   $$\forall i \geq 1, T_i \leq \frac{Q_i}{bw_i} + \Gamma \Rightarrow bw_i\,(t - \Gamma) \leq Q_i.$$

   If we sum these inequalities and if we apply Equation (5.9), we obtain:

   $$(T(N) - b\Delta - 2\Gamma)\, bw_{tot} \leq (t - \Gamma) \sum_{i=1}^{n} bw_i \leq \sum_{i=1}^{n} Q_i = Q. \tag{5.10}$$

   The makespan of an optimal schedule is larger than the time to communicate all data:

   $$\frac{Q}{bw_{tot}} = \frac{1}{bw_{tot}} N \left(\sum_k \pi_k V_{comm}(k)\right) \leq T^*(N). \tag{5.11}$$

   Thanks to Equations (5.10) and (5.11), we can write:

   $$T(n) - b\Delta - 2\Gamma \leq \frac{Q}{bw_{tot}} \leq T^*(N).$$

And then:

$$\frac{T(N)}{T^*(N)} \leq 1 + \frac{b\Delta + 2\Gamma}{T^*(N)}.$$

Thanks again to Equation (5.11), we have $\frac{T(N)}{T^*(N)} \leq 1 + o_{N\to\infty}(1)$. Thus, On-Demand is asymptotically optimal:

$$\lim_{N\to\infty} \frac{\rho(N)}{\rho^*(N)} = 1.$$

2. The master cannot simultaneously serve all workers at full speed: $bw_0 < \sum_{i=1}^{n} bw_i$. Without any loss of generality, we assume that the bandwidth of the master is larger than any other bandwidth: $\forall i \geq 1, bw_0 \geq bw_i$.
   Let $\beta_i(v)$ denote the bandwidth allocated at time v to processor P_i. By definition, we have for all processor P_i and at any time v:

$$0 \leq \beta_i(v) \leq bw_i \text{ and } \sum_{i=1}^{n} \beta_i(v) \leq bw_0.$$

   The first worker achieves its work at time t. At this time, there is no more instances to process. Moreover, just before time t, any other processor P_i may have requested for at most b instances, otherwise its buffer would be overloaded. The master can send all these $(n-1)b$ instances in time $(n-1)\,b\Delta$. So, P_i finishes its work before time $t + (n-1)\,b\Delta + \Gamma$:

$$\forall i \geq 1, T_i \leq t + (n-1)\,b\Delta + \Gamma \Rightarrow T(N) - (n-1)\,b\Delta - \Gamma \leq t. \tag{5.12}$$

   Since Q_i is the total volume of data received by worker P_i, we have:

$$\forall i \geq 1, Q_i = \int_0^{T_i} \beta_i(v)dv.$$

   The instantaneous bandwidth is a non-negative function:

$$\forall i \geq 1, \int_0^{t-\Gamma} \beta_i(v)dv \leq \int_0^{T_i-\Gamma} \beta_i(v)dv \leq \int_0^{T_i} \beta_i(v)dv.$$

   We can sum these inequalities:

$$\int_0^{t-\Gamma} \sum_{i=1}^{n} \beta_i(v)dv = \sum_{i=1}^{n} \int_0^{t-\Gamma} \beta_i(v)dv \leq \sum_{i=1}^{n} Q_i = Q.$$

   From time 0 to time $t-\Gamma$, all processors are receiving data from the master. Thus, the sum of all instantaneous bandwidths corresponds to the whole master bandwidth, even if the transmission rate is variable: $\sum_{i=1}^{n} \beta_i(v) = bw_0$. This leads us to:

$$bw_0\,(t-\Gamma) = \int_{v=0}^{t-\Gamma} \sum_{i=1}^{n} \beta_i(v) \leq Q. \tag{5.13}$$

   The makespan of an optimal schedule is larger than the time to communicate all data:

$$\frac{Q}{bw_0} = \frac{1}{bw_0} N \left(\sum_k \pi_k V_{comm}(k) \right) \leq T^*(N). \tag{5.14}$$

Combining Equations (5.12) and (5.14) to Equation (5.13) gives us:

$$bw_0\left(T(N) - 2\Gamma - (n-1)\,b\Delta\right) \leq bw_0\left(t - \Gamma\right) \leq Q \leq T^*(N)\,bw_0.$$

Similarly to the previous cases, we have:

$$\frac{T(N)}{T^*(N)} \leq 1 + \frac{2\Gamma + (n-1)\,b\Delta}{T^*(N)}.$$

Thanks again to Equation (5.14), we have $\lim_{N\to\infty} T^*(N) = \infty$ and then:

$$\frac{T(N)}{T^*(N)} \leq 1 + o_{N\to\infty}(1).$$

We finally have shown that On-Demand is asymptotically optimal when communications dominate:

$$\lim_{N\to\infty} \frac{\rho(N)}{\rho^*(N)} = 1.$$

Thus, if computations or communications are dominant, the simple On-Demand heuristic with finite buffers is asymptotically optimal.

■

One could think that a weaker condition (for any task, the worst communication time is smaller than its best computation time: $\forall T_k, \frac{V_{comp}(k)}{\max_{1\leq i\leq n}(s_i)} \geq \frac{V_{comm}(k)}{\min_{0\leq i\leq n}(bw_i)}$) is sufficient to ensure the optimality of the On-Demand policy. However, this requires infinite buffers, while actual On-Demand implementations are often limited to a buffer of 2 or 3 instances. If we consider the simple example with a buffer limited to 2 instances, given in Figure 5.5. This weaker condition holds true, but the overlap of communications by computations is not complete: the long reception of an instance of T_1 cannot be started during the long computation of an instance of the same application since the reception buffer is full.

5.4 Experiments

5.4.1 Simulation settings

All algorithms are simulated using the SimGrid framework [31], while the solution of the linear programs is given by GLPK [49]. Each simulation is made of three or four applications, each application being represented by 100, 1,000 or 5,000 instances. First, the communication size is chosen using a uniform probability law over an interval $[\min_{comm}; \max_{comm}]$, such that $\max_{comm}$ is selected in the set $\{\min_{comm}, 1.35\min_{comm}, 1.65\min_{comm}, 2.35\min_{comm}\, 2.65\min_{comm}\}$.

Then, the volume of computations is computed using a *correlation factor* ϕ between 0 and 1: if ϕ is equal to 0, then there is no correlation between the communication size and the volume of computations. On the contrary, if ϕ is equal to 1, then there is a strong correlation between the volume of computations and the communication size of the instance. Let $V_{comm}(i)$ be the communication size of the i-th instance of application T_k, and let $V_{comp}(i)$ be its volume of computations. Since $V_{comm}(i)$ is randomly chosen between $\min^k_{comm}$ and $\max^k_{comm}$, there exists λ, such that $V_{comm}(i) = \lambda\min^k_{comm} + (1-\lambda)\max^k_{comm}$. Taking the correlation factor into account, $V_{comp}(i)$ is randomly chosen between $\left((\phi\lambda + 1 - \phi)\min^k_{comp} + \phi(1-\lambda)\max^k_{comp}\right)$

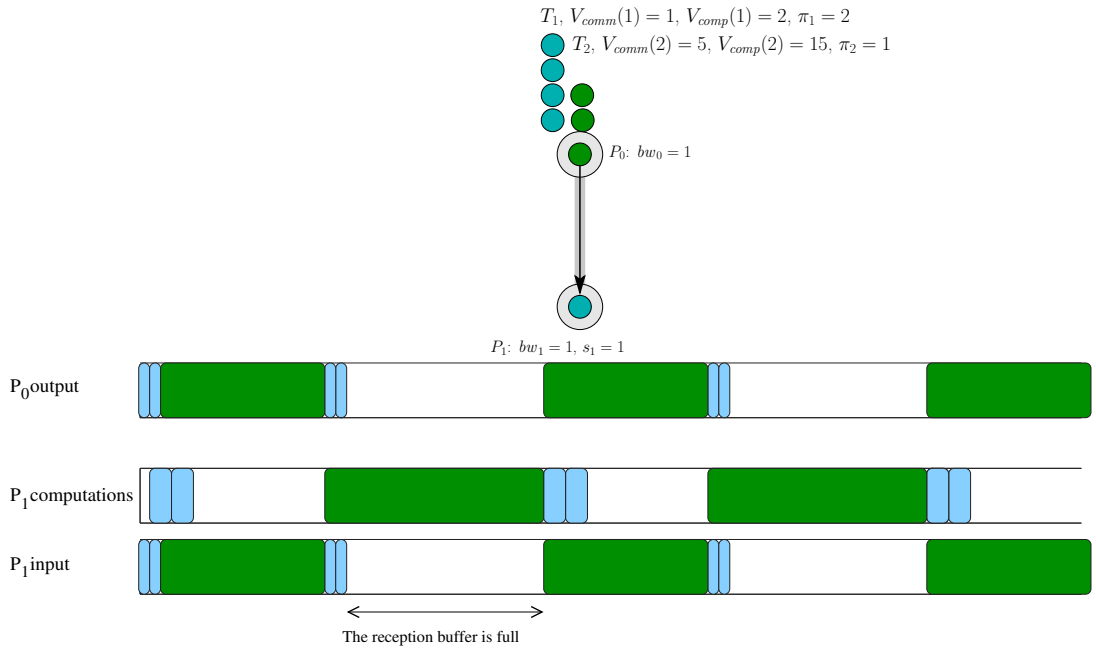


Figure 5.5: Performance loss with a On-Demand scheduler using finite buffers.

and $\left(\phi\lambda \min_{comp}^{k} + (1-\lambda\phi)\max_{comp}^{k}\right)$. Thus, if ϕ is equal to 0, then the communication size is not used to choose the volume of computations. On the contrary, if ϕ is equal to 1, then the computation amount is completely determined by $V_{comm}(i)$.

Finally, instances of all applications are shuffled before the submission of the whole set of instances to the schedule algorithm.

Platforms were constituted of 3, 5, 10, or 15 workers. Many combinations of heuristics, bandwidths, processor speeds, data sizes, and volumes of computations were used, explaining the grand total of more than 540,000 runs.

The average throughput is computed using the number of instances divided by the total computation time (the makespan), with the hope that there are enough instances to dismiss the effects of the initialization and termination phases of steady-state methods. An upper bound of the throughput is computed using the throughput given by the 0.05-approximation, neglecting both initialization and termination phases: by definition of an ε-approximation, the optimal throughput is less than 1.05 times the theoretical throughput of the 0.05-approximation.

Steady-state methods. In our experiments, we denote by LP_SAMP(m, c, d) the heuristic described in Section 5.3.3, splitting the instances following the m method into cd buckets, file sizes being split into c intervals and computation volumes being split into d intervals. m is either GEOM (the geometrical method explained in paragraph 5.3.3), ARITH (the arithmetical one, see paragraph 5.3.3) or REC (the recursive one, explained in paragraph 5.3.3).

Both On-Demand and Round-Robin were tested on all sets. We also used a 0.2- and a 0.05-approximation, as well as LP_SAMP(m, c, d) using 10% of the instances as sample, splitting the values into 1, 2, 4, and 8 buckets. To use the same conditions for all heuristics, these first 10% are ignored by other methods. The approximations are clairvoyant algorithms, while the LP_SAMP(m, c, d) are semi-clairvoyant.

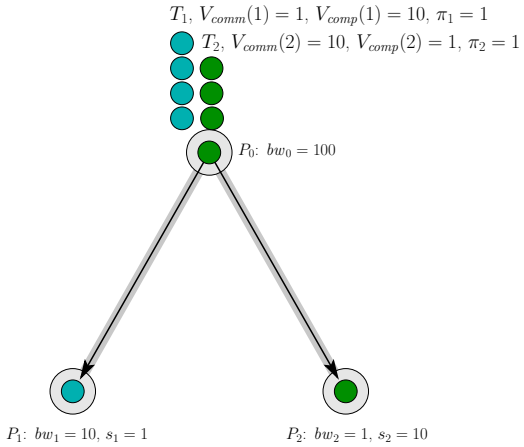


Figure 5.6: Example B: two applications deployed on a two-worker platform.

Running times. Solving linear programs requires complex algorithms such as the Simplex and one can be scared by long running times, but tests with 20 workers and 100 different applications, each with $5,000$ instances lead to a running time of 2 seconds for parsing all data and solving the linear program, while the whole run takes 22 seconds. As soon as linear programs are solved over the rationals, the running times are very small.

5.4.2 Results

Theoretical example. First, we used our framework to run simple situations. Consider the small example explained in [22], with a master, two workers and two applications, as described in Figure 5.6. When a processor is asking for an instance to process, the On-Demand policy does not consider the different volumes of communication or computation. Since all instances are shuffled, any instance has the same probability of being processed by either processor P_1 or processor P_2. Thus, a large number of instances leads to roughly the same number of instances of T_1 and T_2 processed on P_1 and P_2, giving a throughput equal to $2/11 \approx 0.182$. An actual schedule of this situation is shown on Figure 5.7, and results in a poor use of the resources.

On the contrary, the optimal schedule is obtained by placing all instances of T_1 on P_2 and all instances of T_2 on P_1. Our ε-approximation, shown in Figure 5.8, returns this placement and leads to a far better use of the platform with a throughput 1.0. Simulations are in agreement with theoretical numbers: the effective throughput of the first solution is 0.18 instance of each application computed per time unit, while the second one provides an average throughput of 0.999. This is a typical example of situations in which dynamic algorithms perform poorly in regard to static policies.

Mainstream simulations. The previous example demonstrates that clever schedules are required in some configurations to reach a good throughput. However, such situations may be rarely encountered, or even caricatural. Thus, we performed a large number of simulations corresponding to many different contexts. For each set, we normalized the average throughput of each method to the best result among all methods, and to the upper bound of the throughput provided by the 0.05-approximation. We also provide the standard deviation of these normalized values, and their minimum.

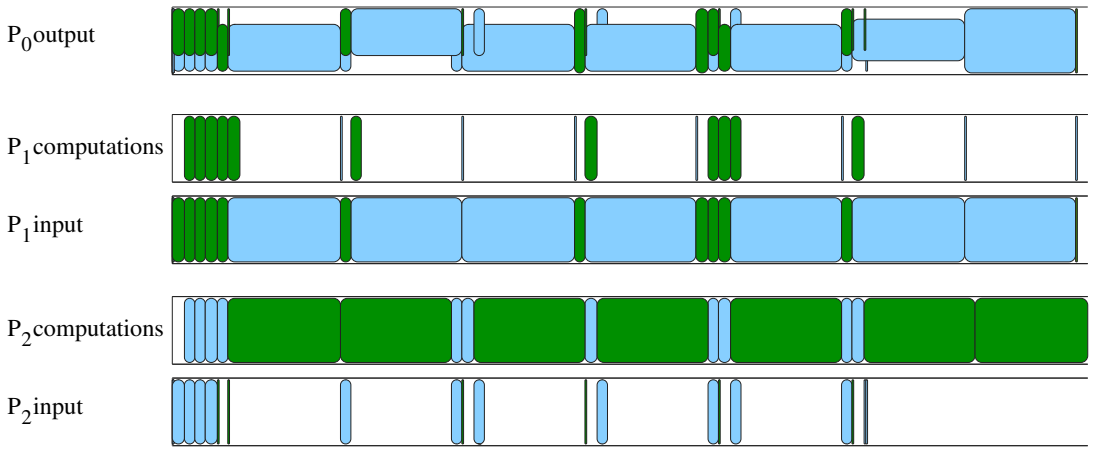


Figure 5.7: Example B: Execution of the first instances using the On-Demand policy.

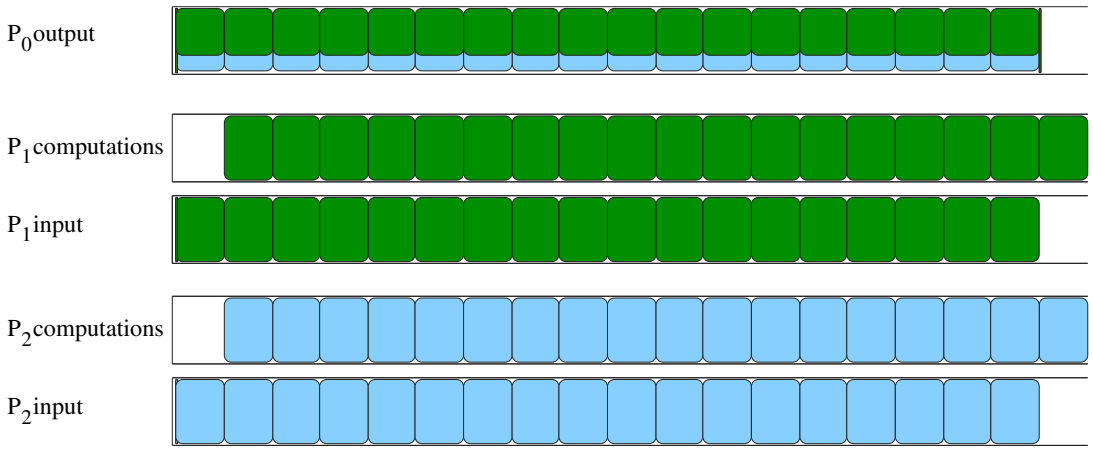


Figure 5.8: Example B: Execution of the first instances using the solution of the linear program.

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.87 ($\sigma = 0.108$, min = 0.638)	0.821 ($\sigma = 0.109$, min = 0.529)
Round-Robin	0.779 ($\sigma = 0.123$, min = 0.443)	0.736 ($\sigma = 0.126$, min = 0.371)
LP_samp(ARITH, 1, 1)	0.971 ($\sigma = 0.0362$, min = 0.692)	0.917 ($\sigma = 0.0651$, min = 0.573)
LP_samp(ARITH, 2, 1)	0.86 ($\sigma = 0.107$, min = 0.226)	0.814 ($\sigma = 0.121$, min = 0.175)
LP_samp(GEOM, 2, 1)	0.875 ($\sigma = 0.106$, min = 0.248)	0.829 ($\sigma = 0.122$, min = 0.2)
LP_samp(REC, 2, 1)	0.868 ($\sigma = 0.106$, min = 0.239)	0.822 ($\sigma = 0.122$, min = 0.197)
LP_samp(ARITH, 2, 2)	0.834 ($\sigma = 0.129$, min = 0.219)	0.791 ($\sigma = 0.144$, min = 0.177)
LP_samp(GEOM, 2, 2)	0.842 ($\sigma = 0.129$, min = 0.213)	0.799 ($\sigma = 0.144$, min = 0.186)
LP_samp(REC, 2, 2)	0.831 ($\sigma = 0.132$, min = 0.0442)	0.788 ($\sigma = 0.147$, min = 0.04)
LP_samp(ARITH, 4, 1)	0.815 ($\sigma = 0.131$, min = 0.167)	0.773 ($\sigma = 0.145$, min = 0.134)
LP_samp(GEOM, 4, 1)	0.819 ($\sigma = 0.13$, min = 0.213)	0.777 ($\sigma = 0.144$, min = 0.183)
LP_samp(REC, 4, 1)	0.81 ($\sigma = 0.131$, min = 0.213)	0.769 ($\sigma = 0.145$, min = 0.186)
LP_samp(ARITH, 4, 4)	0.81 ($\sigma = 0.14$, min = 0.0541)	0.769 ($\sigma = 0.154$, min = 0.0495)
LP_samp(GEOM, 4, 4)	0.812 ($\sigma = 0.139$, min = 0.0896)	0.771 ($\sigma = 0.153$, min = 0.0775)
LP_samp(REC, 4, 4)	0.803 ($\sigma = 0.141$, min = 0.00353)	0.763 ($\sigma = 0.155$, min = 0.00313)
LP_samp(ARITH, 8, 1)	0.791 ($\sigma = 0.137$, min = 0.171)	0.751 ($\sigma = 0.15$, min = 0.137)
LP_samp(GEOM, 8, 1)	0.795 ($\sigma = 0.136$, min = 0.151)	0.754 ($\sigma = 0.149$, min = 0.139)
LP_samp(REC, 8, 1)	0.784 ($\sigma = 0.14$, min = 0.171)	0.744 ($\sigma = 0.153$, min = 0.151)
0.05-approx	0.993 ($\sigma = 0.022$, min = 0.111)	0.937 ($\sigma = 0.0555$, min = 0.097)
0.2-approx	0.985 ($\sigma = 0.0201$, min = 0.178)	0.93 ($\sigma = 0.0513$, min = 0.148)

Table 5.1: Summary of all experiments. Given figures are the average throughput, the standard deviation σ and the minimum throughput (27,360 runs of each heuristic).

A summary of all the results is displayed in Table 5.1. At a first glance, we see that the best solution is given by the 0.05-approximation with an average throughput equal to 99% of the best solution, followed by the 0.2-approximation (98.5%) and LP_samp(m, 1, 1) (97.1%). Splitting into more buckets surprisingly leads to worse results: only 79%. Finally, the On-Demand policy returns better results than the Round-Robin one (83.8% versus 79.7%), but remains worse than the LP_samp(m, 1,1) policies. In the following paragraphs, we present the influence of the most important parameters on the different methods.

Communication-to-computation ratio. Results obtained with several CCRs are given in Tables 5.2, 5.3, 5.4, 5.5, 5.6. We see that the On-Demand returns very good schedules in case of very large (Table 5.6 with $CCR = 20$) CCRs or very small ones (Table 5.2, with $CCR = 0.05$), with a throughput being equal on average to 99% of the best one. They are also very close to the upper bound of the optimal values (0.93%). These good results are in agreement with Theorem 5.1, while its results with CCRs between 0.625 and 1.67 are around 80%.

On the contrary, the Round-Robin policy is worse with such extreme values than with moderate CCRs, reaching on average 71% of the best solution with CCRs equal to 0.05 or 20, compared to the 79.5% obtained with CCRs comprised between 0.625 and 1.67.

Our LP_samp(m, c, d) heuristic is mostly independent of the CCR, presenting very small variations, less than 5%. Our ε-approximation is very slightly affected by extreme CCRs, still offering an average throughput larger than 97% of the best solution, instead of the 98% or the 99%, that are reached with more balanced CCRs. In all cases, results of our methods are close to the optimal, realizing more than 90% of the optimal throughput.

Number of instances. Contrarily to the On-Demand and the Round-Robin policies, the steady-state approach requires large number of instances to be efficient. Thus, we present in

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.993 (σ = 0.00687, min = 0.924)	0.937 (σ = 0.0397, min = 0.728)
Round-Robin	0.716 (σ = 0.101, min = 0.444)	0.676 (σ = 0.104, min = 0.393)
LP_samp(ARITH, 1, 1)	0.97 (σ = 0.0443, min = 0.715)	0.917 (σ = 0.0749, min = 0.631)
LP_samp(ARITH, 2, 1)	0.869 (σ = 0.122, min = 0.281)	0.823 (σ = 0.133, min = 0.238)
LP_samp(GEOM, 2, 1)	0.878 (σ = 0.118, min = 0.268)	0.832 (σ = 0.132, min = 0.221)
LP_samp(REC, 2, 1)	0.847 (σ = 0.116, min = 0.239)	0.803 (σ = 0.131, min = 0.197)
LP_samp(ARITH, 2, 2)	0.853 (σ = 0.152, min = 0.25)	0.809 (σ = 0.162, min = 0.223)
LP_samp(GEOM, 2, 2)	0.86 (σ = 0.148, min = 0.246)	0.816 (σ = 0.16, min = 0.223)
LP_samp(REC, 2, 2)	0.814 (σ = 0.15, min = 0.193)	0.773 (σ = 0.164, min = 0.171)
LP_samp(ARITH, 4, 1)	0.796 (σ = 0.147, min = 0.205)	0.756 (σ = 0.158, min = 0.175)
LP_samp(GEOM, 4, 1)	0.797 (σ = 0.144, min = 0.214)	0.756 (σ = 0.155, min = 0.183)
LP_samp(REC, 4, 1)	0.771 (σ = 0.148, min = 0.213)	0.732 (σ = 0.159, min = 0.194)
LP_samp(ARITH, 4, 4)	0.802 (σ = 0.157, min = 0.193)	0.761 (σ = 0.169, min = 0.171)
LP_samp(GEOM, 4, 4)	0.801 (σ = 0.157, min = 0.193)	0.761 (σ = 0.169, min = 0.171)
LP_samp(REC, 4, 4)	0.771 (σ = 0.162, min = 0.203)	0.733 (σ = 0.173, min = 0.179)
LP_samp(ARITH, 8, 1)	0.759 (σ = 0.153, min = 0.216)	0.721 (σ = 0.163, min = 0.184)
LP_samp(GEOM, 8, 1)	0.76 (σ = 0.151, min = 0.211)	0.722 (σ = 0.162, min = 0.181)
LP_samp(REC, 8, 1)	0.739 (σ = 0.154, min = 0.171)	0.702 (σ = 0.165, min = 0.151)
0.05-approx	0.979 (σ = 0.0392, min = 0.763)	0.926 (σ = 0.0714, min = 0.621)
0.2-approx	0.974 (σ = 0.0376, min = 0.743)	0.92 (σ = 0.0691, min = 0.622)

Table 5.2: Communications / Computations = 0.05 (2,880 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.843 (σ = 0.0852, min = 0.665)	0.789 (σ = 0.0834, min = 0.563)
Round-Robin	0.781 (σ = 0.115, min = 0.482)	0.732 (σ = 0.118, min = 0.384)
LP_samp(ARITH, 1, 1)	0.973 (σ = 0.0312, min = 0.765)	0.912 (σ = 0.0685, min = 0.573)
LP_samp(ARITH, 2, 1)	0.849 (σ = 0.105, min = 0.258)	0.798 (σ = 0.123, min = 0.199)
LP_samp(GEOM, 2, 1)	0.86 (σ = 0.105, min = 0.258)	0.809 (σ = 0.125, min = 0.218)
LP_samp(REC, 2, 1)	0.857 (σ = 0.106, min = 0.264)	0.806 (σ = 0.126, min = 0.201)
LP_samp(ARITH, 2, 2)	0.828 (σ = 0.129, min = 0.257)	0.78 (σ = 0.148, min = 0.211)
LP_samp(GEOM, 2, 2)	0.833 (σ = 0.129, min = 0.226)	0.785 (σ = 0.149, min = 0.186)
LP_samp(REC, 2, 2)	0.829 (σ = 0.131, min = 0.0908)	0.781 (σ = 0.151, min = 0.0814)
LP_samp(ARITH, 4, 1)	0.795 (σ = 0.127, min = 0.261)	0.749 (σ = 0.144, min = 0.199)
LP_samp(GEOM, 4, 1)	0.796 (σ = 0.125, min = 0.25)	0.75 (σ = 0.143, min = 0.205)
LP_samp(REC, 4, 1)	0.79 (σ = 0.127, min = 0.236)	0.745 (σ = 0.144, min = 0.198)
LP_samp(ARITH, 4, 4)	0.803 (σ = 0.141, min = 0.111)	0.758 (σ = 0.159, min = 0.0974)
LP_samp(GEOM, 4, 4)	0.804 (σ = 0.139, min = 0.0896)	0.758 (σ = 0.157, min = 0.0775)
LP_samp(REC, 4, 4)	0.798 (σ = 0.14, min = 0.148)	0.753 (σ = 0.158, min = 0.128)
LP_samp(ARITH, 8, 1)	0.766 (σ = 0.131, min = 0.219)	0.722 (σ = 0.147, min = 0.188)
LP_samp(GEOM, 8, 1)	0.77 (σ = 0.131, min = 0.239)	0.725 (σ = 0.147, min = 0.2)
LP_samp(REC, 8, 1)	0.761 (σ = 0.133, min = 0.22)	0.718 (σ = 0.149, min = 0.182)
0.05-approx	0.993 (σ = 0.0241, min = 0.111)	0.931 (σ = 0.0632, min = 0.097)
0.2-approx	0.985 (σ = 0.0203, min = 0.178)	0.923 (σ = 0.0583, min = 0.148)

Table 5.3: Communications / Computations = 0.625 (7,200 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.81 ($\sigma = 0.115$, min = 0.638)	0.763 ($\sigma = 0.115$, min = 0.529)
Round-Robin	0.81 ($\sigma = 0.12$, min = 0.51)	0.764 ($\sigma = 0.125$, min = 0.414)
LP_samp(ARITH, 1, 1)	0.958 ($\sigma = 0.0419$, min = 0.75)	0.903 ($\sigma = 0.0686$, min = 0.578)
LP_samp(ARITH, 2, 1)	0.85 ($\sigma = 0.104$, min = 0.226)	0.804 ($\sigma = 0.12$, min = 0.175)
LP_samp(GEOM, 2, 1)	0.866 ($\sigma = 0.103$, min = 0.248)	0.819 ($\sigma = 0.121$, min = 0.2)
LP_samp(REC, 2, 1)	0.865 ($\sigma = 0.102$, min = 0.299)	0.818 ($\sigma = 0.12$, min = 0.244)
LP_samp(ARITH, 2, 2)	0.832 ($\sigma = 0.122$, min = 0.241)	0.788 ($\sigma = 0.139$, min = 0.185)
LP_samp(GEOM, 2, 2)	0.841 ($\sigma = 0.122$, min = 0.246)	0.796 ($\sigma = 0.139$, min = 0.191)
LP_samp(REC, 2, 2)	0.835 ($\sigma = 0.125$, min = 0.0442)	0.792 ($\sigma = 0.142$, min = 0.04)
LP_samp(ARITH, 4, 1)	0.812 ($\sigma = 0.126$, min = 0.167)	0.77 ($\sigma = 0.142$, min = 0.134)
LP_samp(GEOM, 4, 1)	0.815 ($\sigma = 0.124$, min = 0.226)	0.772 ($\sigma = 0.14$, min = 0.183)
LP_samp(REC, 4, 1)	0.81 ($\sigma = 0.124$, min = 0.239)	0.767 ($\sigma = 0.14$, min = 0.186)
LP_samp(ARITH, 4, 4)	0.817 ($\sigma = 0.133$, min = 0.0541)	0.775 ($\sigma = 0.149$, min = 0.0495)
LP_samp(GEOM, 4, 4)	0.819 ($\sigma = 0.132$, min = 0.265)	0.776 ($\sigma = 0.148$, min = 0.212)
LP_samp(REC, 4, 4)	0.814 ($\sigma = 0.134$, min = 0.00353)	0.771 ($\sigma = 0.15$, min = 0.00313)
LP_samp(ARITH, 8, 1)	0.792 ($\sigma = 0.13$, min = 0.171)	0.751 ($\sigma = 0.145$, min = 0.137)
LP_samp(GEOM, 8, 1)	0.794 ($\sigma = 0.129$, min = 0.276)	0.752 ($\sigma = 0.144$, min = 0.215)
LP_samp(REC, 8, 1)	0.784 ($\sigma = 0.132$, min = 0.222)	0.743 ($\sigma = 0.146$, min = 0.198)
0.05-approx	0.995 ($\sigma = 0.0128$, min = 0.807)	0.938 ($\sigma = 0.0528$, min = 0.647)
0.2-approx	0.985 ($\sigma = 0.0166$, min = 0.289)	0.929 ($\sigma = 0.0499$, min = 0.267)

Table 5.4: Communications / Computations = 1 (7,200 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.86 ($\sigma = 0.0897$, min = 0.67)	0.817 ($\sigma = 0.0929$, min = 0.586)
Round-Robin	0.796 ($\sigma = 0.135$, min = 0.446)	0.757 ($\sigma = 0.137$, min = 0.371)
LP_samp(ARITH, 1, 1)	0.98 ($\sigma = 0.0258$, min = 0.812)	0.932 ($\sigma = 0.0511$, min = 0.649)
LP_samp(ARITH, 2, 1)	0.877 ($\sigma = 0.101$, min = 0.237)	0.835 ($\sigma = 0.113$, min = 0.225)
LP_samp(GEOM, 2, 1)	0.891 ($\sigma = 0.102$, min = 0.26)	0.849 ($\sigma = 0.116$, min = 0.23)
LP_samp(REC, 2, 1)	0.887 ($\sigma = 0.102$, min = 0.258)	0.845 ($\sigma = 0.115$, min = 0.213)
LP_samp(ARITH, 2, 2)	0.841 ($\sigma = 0.121$, min = 0.219)	0.803 ($\sigma = 0.134$, min = 0.177)
LP_samp(GEOM, 2, 2)	0.848 ($\sigma = 0.123$, min = 0.213)	0.809 ($\sigma = 0.135$, min = 0.202)
LP_samp(REC, 2, 2)	0.842 ($\sigma = 0.126$, min = 0.213)	0.804 ($\sigma = 0.138$, min = 0.202)
LP_samp(ARITH, 4, 1)	0.844 ($\sigma = 0.123$, min = 0.201)	0.805 ($\sigma = 0.135$, min = 0.189)
LP_samp(GEOM, 4, 1)	0.846 ($\sigma = 0.124$, min = 0.213)	0.807 ($\sigma = 0.136$, min = 0.202)
LP_samp(REC, 4, 1)	0.843 ($\sigma = 0.123$, min = 0.213)	0.804 ($\sigma = 0.136$, min = 0.202)
LP_samp(ARITH, 4, 4)	0.821 ($\sigma = 0.133$, min = 0.179)	0.784 ($\sigma = 0.144$, min = 0.168)
LP_samp(GEOM, 4, 4)	0.824 ($\sigma = 0.133$, min = 0.236)	0.787 ($\sigma = 0.145$, min = 0.209)
LP_samp(REC, 4, 4)	0.818 ($\sigma = 0.135$, min = 0.225)	0.781 ($\sigma = 0.146$, min = 0.209)
LP_samp(ARITH, 8, 1)	0.826 ($\sigma = 0.13$, min = 0.213)	0.789 ($\sigma = 0.142$, min = 0.187)
LP_samp(GEOM, 8, 1)	0.829 ($\sigma = 0.129$, min = 0.151)	0.791 ($\sigma = 0.141$, min = 0.139)
LP_samp(REC, 8, 1)	0.821 ($\sigma = 0.135$, min = 0.227)	0.784 ($\sigma = 0.146$, min = 0.187)
0.05-approx	0.994 ($\sigma = 0.016$, min = 0.15)	0.945 ($\sigma = 0.0437$, min = 0.136)
0.2-approx	0.988 ($\sigma = 0.0119$, min = 0.702)	0.939 ($\sigma = 0.038$, min = 0.67)

Table 5.5: Communications / Computations = 1.67 (7,200 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.987 (σ = 0.0162, min = 0.845)	0.938 (σ = 0.0416, min = 0.695)
Round-Robin	0.719 (σ = 0.0965, min = 0.443)	0.684 (σ = 0.099, min = 0.376)
LP_samp(ARITH, 1, 1)	0.977 (σ = 0.0364, min = 0.692)	0.93 (σ = 0.0586, min = 0.573)
LP_samp(ARITH, 2, 1)	0.862 (σ = 0.108, min = 0.291)	0.821 (σ = 0.118, min = 0.248)
LP_samp(GEOM, 2, 1)	0.894 (σ = 0.105, min = 0.381)	0.851 (σ = 0.116, min = 0.309)
LP_samp(REC, 2, 1)	0.874 (σ = 0.109, min = 0.291)	0.833 (σ = 0.119, min = 0.248)
LP_samp(ARITH, 2, 2)	0.812 (σ = 0.139, min = 0.268)	0.775 (σ = 0.149, min = 0.237)
LP_samp(GEOM, 2, 2)	0.835 (σ = 0.137, min = 0.269)	0.797 (σ = 0.146, min = 0.235)
LP_samp(REC, 2, 2)	0.812 (σ = 0.145, min = 0.207)	0.776 (σ = 0.154, min = 0.175)
LP_samp(ARITH, 4, 1)	0.82 (σ = 0.14, min = 0.241)	0.783 (σ = 0.149, min = 0.207)
LP_samp(GEOM, 4, 1)	0.838 (σ = 0.138, min = 0.217)	0.8 (σ = 0.147, min = 0.187)
LP_samp(REC, 4, 1)	0.818 (σ = 0.139, min = 0.217)	0.781 (σ = 0.149, min = 0.187)
LP_samp(ARITH, 4, 4)	0.79 (σ = 0.148, min = 0.221)	0.755 (σ = 0.157, min = 0.19)
LP_samp(GEOM, 4, 4)	0.796 (σ = 0.148, min = 0.233)	0.76 (σ = 0.157, min = 0.21)
LP_samp(REC, 4, 4)	0.785 (σ = 0.151, min = 0.233)	0.75 (σ = 0.16, min = 0.21)
LP_samp(ARITH, 8, 1)	0.799 (σ = 0.148, min = 0.221)	0.763 (σ = 0.157, min = 0.19)
LP_samp(GEOM, 8, 1)	0.81 (σ = 0.144, min = 0.278)	0.773 (σ = 0.153, min = 0.23)
LP_samp(REC, 8, 1)	0.795 (σ = 0.147, min = 0.223)	0.759 (σ = 0.156, min = 0.191)
0.05-approx	0.993 (σ = 0.0194, min = 0.806)	0.944 (σ = 0.0449, min = 0.692)
0.2-approx	0.986 (σ = 0.0142, min = 0.883)	0.937 (σ = 0.038, min = 0.754)

Table 5.6: Communications / Computations = 20 (2,880 simulations).

Tables 5.7, 5.8, and 5.9 the results of the different methods with respectively 100, 1,000 and 5,000 instances of each application. Table 5.7 clearly shows that if 100 instances are often enough to reach good throughputs with a steady-state approach (on average 98% of the best solution for the ε-approximation, between 68% and 75% for the LP_samp(m, c, d) heuristics), some cases are really tough: the minimum throughput of the ε-approximation can be as low as 11% of the best solution, while the LP_samp(m, 1, 1) has a worst case equal to 69%. Splitting into more buckets returns worse solutions, and the situation tends to be catastrophic when buckets are formed using both communications and computations: LP_samp(m, 4, 4) has some worst cases reaching less than 10% of the best solution. There are two reasons to these poor results: the small size of the sample (only 10 instances of each application), which returns bad indications on the distribution of communication sizes and volumes of computations, and the small numbers of effectively processed instances. However, 100 instances are enough to be quite close to the upper bound on the optimal, since our ε-approximation realizes on average more than 87% of the optimal throughput, while our LP_samp(ARITH, 1, 1) realizes around 85% of this bound.

Using 1,000 (Table 5.8) or 5,000 instances (Table 5.9) significantly increases the average results of LP_samp(m, c, d) using several buckets. Worst cases are also better: the 0.05-approximation has a throughput at least equal to 80% of the best one, while the LP_samp(m, 1, 1) methods offer at least 83% of the best solution. This situation is similar for all methods using a steady-state approach. 1,000 instances are enough to almost reach the optimal, the performance being around 95% of the optimal, both for our approximation and our LP_samp(() m, 1, 1) heuristic. With 5,000 instances, the situation is even better, since our 0.05-approximation offers an average throughput equal to 97% of the upper bound, while our LP_samp(m, 1, 1) realizes more than 96% of the optimal. Moreover, splitting into more buckets is more efficient as the number of instances: on average around 90% of the best solution with 5,000 instances, instead of around 80% with 1,000 instances, and 70% with 100 instances. However, it is still less efficient than using a single bucket.

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.879 ($\sigma = 0.101$, min = 0.64)	0.788 ($\sigma = 0.104$, min = 0.529)
Round-Robin	0.781 ($\sigma = 0.119$, min = 0.443)	0.702 ($\sigma = 0.124$, min = 0.371)
LP_samp(ARITH, 1, 1)	0.951 ($\sigma = 0.0448$, min = 0.692)	0.853 ($\sigma = 0.0691$, min = 0.573)
LP_samp(ARITH, 2, 1)	0.788 ($\sigma = 0.13$, min = 0.226)	0.708 ($\sigma = 0.132$, min = 0.175)
LP_samp(GEOM, 2, 1)	0.793 ($\sigma = 0.127$, min = 0.248)	0.713 ($\sigma = 0.129$, min = 0.2)
LP_samp(REC, 2, 1)	0.786 ($\sigma = 0.127$, min = 0.239)	0.706 ($\sigma = 0.129$, min = 0.197)
LP_samp(ARITH, 2, 2)	0.731 ($\sigma = 0.146$, min = 0.219)	0.658 ($\sigma = 0.15$, min = 0.177)
LP_samp(GEOM, 2, 2)	0.734 ($\sigma = 0.145$, min = 0.213)	0.661 ($\sigma = 0.148$, min = 0.186)
LP_samp(REC, 2, 2)	0.717 ($\sigma = 0.146$, min = 0.0442)	0.646 ($\sigma = 0.15$, min = 0.04)
LP_samp(ARITH, 4, 1)	0.715 ($\sigma = 0.144$, min = 0.167)	0.644 ($\sigma = 0.148$, min = 0.134)
LP_samp(GEOM, 4, 1)	0.718 ($\sigma = 0.142$, min = 0.213)	0.647 ($\sigma = 0.146$, min = 0.183)
LP_samp(REC, 4, 1)	0.709 ($\sigma = 0.143$, min = 0.213)	0.639 ($\sigma = 0.148$, min = 0.186)
LP_samp(ARITH, 4, 4)	0.697 ($\sigma = 0.148$, min = 0.0541)	0.628 ($\sigma = 0.153$, min = 0.0495)
LP_samp(GEOM, 4, 4)	0.698 ($\sigma = 0.147$, min = 0.0896)	0.629 ($\sigma = 0.152$, min = 0.0775)
LP_samp(REC, 4, 4)	0.687 ($\sigma = 0.146$, min = 0.00353)	0.62 ($\sigma = 0.151$, min = 0.00313)
LP_samp(ARITH, 8, 1)	0.694 ($\sigma = 0.146$, min = 0.171)	0.626 ($\sigma = 0.15$, min = 0.137)
LP_samp(GEOM, 8, 1)	0.696 ($\sigma = 0.144$, min = 0.151)	0.627 ($\sigma = 0.149$, min = 0.139)
LP_samp(REC, 8, 1)	0.684 ($\sigma = 0.147$, min = 0.171)	0.617 ($\sigma = 0.152$, min = 0.151)
0.05-approx	0.98 ($\sigma = 0.034$, min = 0.111)	0.879 ($\sigma = 0.0618$, min = 0.097)
0.2-approx	0.98 ($\sigma = 0.0308$, min = 0.178)	0.878 ($\sigma = 0.0596$, min = 0.148)

Table 5.7: Summary of results, with 100 instances of each application (9,120 simulations).

Compared to the best solution, the throughput reached on average by the On-Demand and the Round-Robin policies is not related to the number of instances. On average, On-Demand has a throughput equal to respectively 88%, 87% and 86% with 100, $1,000$ and $5,000$ instances. The corresponding average throughputs of Round-Robin are equal to 78%, 78% and 77%. Even their worst cases are independent of the number of instances, always being around 64% for On-Demand and 45% for Round-Robin.

The upper bound on the reachable throughput provided by the ε-approximation is more precise as the number of instances increases. With 100 instances of each application, On-Demand reaches on average 79% of this bound, compared to the 88% of the ε-approximation. As expected due to the increased precision of the upper bound, the results are better when dealing with $1,000$ or $5,000$ instances: On-Demand reaches 83% of the upper bound. Similarly, Round-Robin has an average throughput equal to 70% of the upper bound with 100 instances, and 75% with at least $1,000$ instances.

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.866 (σ = 0.111, min = 0.64)	0.834 (σ = 0.108, min = 0.605)
Round-Robin	0.778 (σ = 0.124, min = 0.499)	0.749 (σ = 0.122, min = 0.477)
LP_samp(ARITH, 1, 1)	0.977 (σ = 0.0269, min = 0.819)	0.941 (σ = 0.031, min = 0.763)
LP_samp(ARITH, 2, 1)	0.881 (σ = 0.0753, min = 0.554)	0.849 (σ = 0.0762, min = 0.518)
LP_samp(GEOM, 2, 1)	0.897 (σ = 0.0668, min = 0.604)	0.864 (σ = 0.0678, min = 0.569)
LP_samp(REC, 2, 1)	0.889 (σ = 0.0661, min = 0.579)	0.856 (σ = 0.0674, min = 0.555)
LP_samp(ARITH, 2, 2)	0.856 (σ = 0.0905, min = 0.433)	0.825 (σ = 0.0911, min = 0.409)
LP_samp(GEOM, 2, 2)	0.864 (σ = 0.0843, min = 0.394)	0.833 (σ = 0.0848, min = 0.37)
LP_samp(REC, 2, 2)	0.852 (σ = 0.0857, min = 0.465)	0.821 (σ = 0.0863, min = 0.446)
LP_samp(ARITH, 4, 1)	0.836 (σ = 0.0921, min = 0.403)	0.806 (σ = 0.0933, min = 0.379)
LP_samp(GEOM, 4, 1)	0.839 (σ = 0.0916, min = 0.413)	0.809 (σ = 0.0926, min = 0.39)
LP_samp(REC, 4, 1)	0.828 (σ = 0.0933, min = 0.404)	0.798 (σ = 0.0943, min = 0.379)
LP_samp(ARITH, 4, 4)	0.826 (σ = 0.103, min = 0.219)	0.796 (σ = 0.103, min = 0.21)
LP_samp(GEOM, 4, 4)	0.828 (σ = 0.101, min = 0.427)	0.798 (σ = 0.101, min = 0.405)
LP_samp(REC, 4, 4)	0.819 (σ = 0.106, min = 0.384)	0.79 (σ = 0.106, min = 0.369)
LP_samp(ARITH, 8, 1)	0.804 (σ = 0.107, min = 0.346)	0.775 (σ = 0.107, min = 0.327)
LP_samp(GEOM, 8, 1)	0.808 (σ = 0.105, min = 0.347)	0.779 (σ = 0.105, min = 0.325)
LP_samp(REC, 8, 1)	0.797 (σ = 0.109, min = 0.347)	0.768 (σ = 0.11, min = 0.328)
0.05-approx	0.998 (σ = 0.00744, min = 0.803)	0.961 (σ = 0.013, min = 0.751)
0.2-approx	0.988 (σ = 0.0109, min = 0.702)	0.952 (σ = 0.0146, min = 0.676)

Table 5.8: Summary of results, with 1,000 instances of each application (9,120 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.863 ($\sigma = 0.112$, min = 0.638)	0.84 ($\sigma = 0.109$, min = 0.618)
Round-Robin	0.779 ($\sigma = 0.127$, min = 0.509)	0.758 ($\sigma = 0.125$, min = 0.494)
LP_samp(ARITH, 1, 1)	0.984 ($\sigma = 0.0241$, min = 0.83)	0.958 ($\sigma = 0.0245$, min = 0.804)
LP_samp(ARITH, 2, 1)	0.911 ($\sigma = 0.0575$, min = 0.691)	0.886 ($\sigma = 0.0574$, min = 0.672)
LP_samp(GEOM, 2, 1)	0.935 ($\sigma = 0.0492$, min = 0.734)	0.91 ($\sigma = 0.0489$, min = 0.714)
LP_samp(REC, 2, 1)	0.928 ($\sigma = 0.0496$, min = 0.716)	0.903 ($\sigma = 0.0496$, min = 0.694)
LP_samp(ARITH, 2, 2)	0.915 ($\sigma = 0.055$, min = 0.626)	0.89 ($\sigma = 0.055$, min = 0.606)
LP_samp(GEOM, 2, 2)	0.928 ($\sigma = 0.0456$, min = 0.61)	0.903 ($\sigma = 0.0454$, min = 0.592)
LP_samp(REC, 2, 2)	0.922 ($\sigma = 0.0471$, min = 0.124)	0.897 ($\sigma = 0.0472$, min = 0.12)
LP_samp(ARITH, 4, 1)	0.894 ($\sigma = 0.0725$, min = 0.475)	0.87 ($\sigma = 0.072$, min = 0.459)
LP_samp(GEOM, 4, 1)	0.899 ($\sigma = 0.0713$, min = 0.608)	0.875 ($\sigma = 0.0706$, min = 0.588)
LP_samp(REC, 4, 1)	0.893 ($\sigma = 0.0699$, min = 0.586)	0.869 ($\sigma = 0.0694$, min = 0.563)
LP_samp(ARITH, 4, 4)	0.908 ($\sigma = 0.058$, min = 0.511)	0.884 ($\sigma = 0.058$, min = 0.495)
LP_samp(GEOM, 4, 4)	0.91 ($\sigma = 0.0566$, min = 0.574)	0.886 ($\sigma = 0.0565$, min = 0.559)
LP_samp(REC, 4, 4)	0.903 ($\sigma = 0.0628$, min = 0.354)	0.878 ($\sigma = 0.0625$, min = 0.344)
LP_samp(ARITH, 8, 1)	0.877 ($\sigma = 0.0825$, min = 0.436)	0.853 ($\sigma = 0.0817$, min = 0.424)
LP_samp(GEOM, 8, 1)	0.88 ($\sigma = 0.0818$, min = 0.504)	0.857 ($\sigma = 0.081$, min = 0.485)
LP_samp(REC, 8, 1)	0.872 ($\sigma = 0.0847$, min = 0.476)	0.849 ($\sigma = 0.0839$, min = 0.461)
0.05-approx	0.999 ($\sigma = 0.00442$, min = 0.807)	0.972 ($\sigma = 0.00563$, min = 0.787)
0.2-approx	0.986 ($\sigma = 0.0102$, min = 0.709)	0.959 ($\sigma = 0.0109$, min = 0.69)

Table 5.9: Summary of results, with 5,000 instances of each application (9,120 simulations).

Range of random variables. As said in Section 5.4.1, communication and computation volumes are determined following a uniform probability law. These volumes may be constant, or, on the contrary, may vary over a large range of values. The standard deviation is not suited to measure these variations, since large applications may have a large standard deviation even if their sizes are almost constant. Thus, if we consider the bounds $\min^k_{comm}$, $\max^k_{comm}$, $\min^k_{comp}$ and $\max^k_{comp}$ on the volumes of an application T_k, we define the deviation of this application as:

$$\nu^k = \frac{\max^k_{comm} - \min^k_{comm}}{\min^k_{comm}} + \frac{\max^k_{comp} - \min^k_{comp}}{\min^k_{comp}}.$$

In Tables 5.10, 5.11, 5.12 and 5.13, the deviation is the same for all applications, even if they have different sizes, and is equal in these tables to respectively 0.05, 0.75, 1.1 and 10.7. When instances are almost constant ($\nu^k = 0.05$), steady-state methods ensure very good throughputs and offering in almost all cases the best solution. In this case, steady-state methods are extremely close to the upper bound on the optimal: 97.5% for the ε-approximation, between 90% and 97.5% for the heuristics . Even with large deviations, up to 10.7, using a single bucket for all instances remains the most efficient solution among all LP_samp(m, c, d) methods. Our steady-state heuristics are almost unaffected by large deviations. Due to its dynamicity, On-Demand is able to easily cope with large deviations, giving higher throughputs than our LP_samp(m, c, d) by a short hand. Finally, our ε-approximation still returns the highest throughputs.

Splitting methods. Table 5.9 provides a good overview of the results of the methods using our LP_samp(m, c, d) heuristic. The choice of method to gather instances into buckets has a little importance, but the geometrical remains slightly better than the arithmetical one, itself better than the recursive one. In this table, the difference between the average results of these three methods is less than 2%.

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.986 (σ = 0.00384, min = 0.975)	0.962 (σ = 0.00371, min = 0.953)
Round-Robin	0.706 (σ = 0.0948, min = 0.562)	0.688 (σ = 0.0932, min = 0.547)
LP_samp(ARITH, 1, 1)	1 (σ = 0.00028, min = 0.997)	0.975 (σ = 0.00178, min = 0.97)
LP_samp(ARITH, 2, 1)	1 (σ = 0.00028, min = 0.997)	0.975 (σ = 0.00178, min = 0.97)
LP_samp(GEOM, 2, 1)	1 (σ = 0.00028, min = 0.997)	0.975 (σ = 0.00178, min = 0.97)
LP_samp(REC, 2, 1)	0.953 (σ = 0.0248, min = 0.859)	0.93 (σ = 0.0248, min = 0.837)
LP_samp(ARITH, 2, 2)	1 (σ = 0.00028, min = 0.997)	0.975 (σ = 0.00178, min = 0.97)
LP_samp(GEOM, 2, 2)	1 (σ = 0.00028, min = 0.997)	0.975 (σ = 0.00178, min = 0.97)
LP_samp(REC, 2, 2)	0.927 (σ = 0.0443, min = 0.786)	0.904 (σ = 0.0441, min = 0.764)
LP_samp(ARITH, 4, 1)	0.954 (σ = 0.0343, min = 0.783)	0.93 (σ = 0.0341, min = 0.761)
LP_samp(GEOM, 4, 1)	0.952 (σ = 0.0333, min = 0.806)	0.928 (σ = 0.0332, min = 0.784)
LP_samp(REC, 4, 1)	0.92 (σ = 0.0439, min = 0.769)	0.897 (σ = 0.0438, min = 0.749)
LP_samp(ARITH, 4, 4)	0.939 (σ = 0.0438, min = 0.767)	0.916 (σ = 0.0436, min = 0.745)
LP_samp(GEOM, 4, 4)	0.938 (σ = 0.043, min = 0.772)	0.915 (σ = 0.0429, min = 0.75)
LP_samp(REC, 4, 4)	0.873 (σ = 0.0794, min = 0.657)	0.851 (σ = 0.0786, min = 0.638)
LP_samp(ARITH, 8, 1)	0.932 (σ = 0.0362, min = 0.788)	0.909 (σ = 0.0362, min = 0.766)
LP_samp(GEOM, 8, 1)	0.933 (σ = 0.0359, min = 0.811)	0.91 (σ = 0.036, min = 0.789)
LP_samp(REC, 8, 1)	0.894 (σ = 0.0606, min = 0.688)	0.871 (σ = 0.0602, min = 0.669)
0.05-approx	1 (σ = 0.000292, min = 0.998)	0.975 (σ = 0.00179, min = 0.97)
0.2-approx	1 (σ = 0.000292, min = 0.998)	0.975 (σ = 0.00179, min = 0.97)

Table 5.10: $\nu^k = 0.05$, with 5,000 instances of each application (240 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.829 (σ = 0.103, min = 0.661)	0.807 (σ = 0.101, min = 0.642)
Round-Robin	0.791 (σ = 0.135, min = 0.534)	0.77 (σ = 0.132, min = 0.519)
LP_samp(ARITH, 1, 1)	0.98 (σ = 0.0269, min = 0.874)	0.954 (σ = 0.0269, min = 0.846)
LP_samp(ARITH, 2, 1)	0.909 (σ = 0.0534, min = 0.693)	0.884 (σ = 0.0528, min = 0.673)
LP_samp(GEOM, 2, 1)	0.932 (σ = 0.0469, min = 0.749)	0.907 (σ = 0.0466, min = 0.727)
LP_samp(REC, 2, 1)	0.929 (σ = 0.0478, min = 0.742)	0.904 (σ = 0.0474, min = 0.72)
LP_samp(ARITH, 2, 2)	0.913 (σ = 0.0534, min = 0.626)	0.889 (σ = 0.053, min = 0.606)
LP_samp(GEOM, 2, 2)	0.925 (σ = 0.0451, min = 0.69)	0.9 (σ = 0.0449, min = 0.669)
LP_samp(REC, 2, 2)	0.924 (σ = 0.0456, min = 0.688)	0.899 (σ = 0.0453, min = 0.668)
LP_samp(ARITH, 4, 1)	0.892 (σ = 0.0686, min = 0.665)	0.869 (σ = 0.0678, min = 0.645)
LP_samp(GEOM, 4, 1)	0.895 (σ = 0.0684, min = 0.683)	0.871 (σ = 0.0677, min = 0.659)
LP_samp(REC, 4, 1)	0.896 (σ = 0.0674, min = 0.691)	0.872 (σ = 0.0667, min = 0.669)
LP_samp(ARITH, 4, 4)	0.91 (σ = 0.0572, min = 0.69)	0.886 (σ = 0.0568, min = 0.665)
LP_samp(GEOM, 4, 4)	0.912 (σ = 0.0562, min = 0.598)	0.887 (σ = 0.0557, min = 0.581)
LP_samp(REC, 4, 4)	0.907 (σ = 0.0594, min = 0.683)	0.883 (σ = 0.0589, min = 0.658)
LP_samp(ARITH, 8, 1)	0.877 (σ = 0.0796, min = 0.568)	0.853 (σ = 0.0786, min = 0.547)
LP_samp(GEOM, 8, 1)	0.882 (σ = 0.0769, min = 0.599)	0.859 (σ = 0.0759, min = 0.578)
LP_samp(REC, 8, 1)	0.873 (σ = 0.0815, min = 0.565)	0.85 (σ = 0.0805, min = 0.548)
0.05-approx	0.999 (σ = 0.00589, min = 0.807)	0.973 (σ = 0.00607, min = 0.787)
0.2-approx	0.987 (σ = 0.00904, min = 0.955)	0.96 (σ = 0.01, min = 0.925)

Table 5.11: $\nu^k = 0.75$, with 5,000 instances of each application (1,440 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.846 ($\sigma = 0.115$, min = 0.638)	0.823 ($\sigma = 0.112$, min = 0.618)
Round-Robin	0.784 ($\sigma = 0.128$, min = 0.509)	0.762 ($\sigma = 0.125$, min = 0.494)
LP_samp(ARITH, 1, 1)	0.977 ($\sigma = 0.0306$, min = 0.83)	0.95 ($\sigma = 0.0308$, min = 0.804)
LP_samp(ARITH, 2, 1)	0.884 ($\sigma = 0.0615$, min = 0.702)	0.86 ($\sigma = 0.0613$, min = 0.679)
LP_samp(GEOM, 2, 1)	0.923 ($\sigma = 0.0581$, min = 0.764)	0.898 ($\sigma = 0.0581$, min = 0.738)
LP_samp(REC, 2, 1)	0.908 ($\sigma = 0.0592$, min = 0.718)	0.883 ($\sigma = 0.0592$, min = 0.694)
LP_samp(ARITH, 2, 2)	0.902 ($\sigma = 0.0535$, min = 0.68)	0.877 ($\sigma = 0.0533$, min = 0.662)
LP_samp(GEOM, 2, 2)	0.925 ($\sigma = 0.0421$, min = 0.734)	0.899 ($\sigma = 0.042$, min = 0.711)
LP_samp(REC, 2, 2)	0.918 ($\sigma = 0.046$, min = 0.696)	0.892 ($\sigma = 0.0459$, min = 0.676)
LP_samp(ARITH, 4, 1)	0.862 ($\sigma = 0.0894$, min = 0.561)	0.838 ($\sigma = 0.0886$, min = 0.541)
LP_samp(GEOM, 4, 1)	0.873 ($\sigma = 0.0881$, min = 0.608)	0.849 ($\sigma = 0.0874$, min = 0.588)
LP_samp(REC, 4, 1)	0.87 ($\sigma = 0.0869$, min = 0.61)	0.846 ($\sigma = 0.0862$, min = 0.589)
LP_samp(ARITH, 4, 4)	0.901 ($\sigma = 0.0603$, min = 0.511)	0.876 ($\sigma = 0.06$, min = 0.495)
LP_samp(GEOM, 4, 4)	0.906 ($\sigma = 0.0579$, min = 0.678)	0.88 ($\sigma = 0.0578$, min = 0.659)
LP_samp(REC, 4, 4)	0.901 ($\sigma = 0.0621$, min = 0.568)	0.876 ($\sigma = 0.0618$, min = 0.554)
LP_samp(ARITH, 8, 1)	0.848 ($\sigma = 0.101$, min = 0.436)	0.824 ($\sigma = 0.0999$, min = 0.424)
LP_samp(GEOM, 8, 1)	0.85 ($\sigma = 0.103$, min = 0.539)	0.827 ($\sigma = 0.102$, min = 0.517)
LP_samp(REC, 8, 1)	0.843 ($\sigma = 0.104$, min = 0.555)	0.82 ($\sigma = 0.103$, min = 0.535)
0.05-approx	1 ($\sigma = 0.00144$, min = 0.984)	0.972 ($\sigma = 0.00377$, min = 0.946)
0.2-approx	0.987 ($\sigma = 0.0108$, min = 0.709)	0.959 ($\sigma = 0.0111$, min = 0.69)

Table 5.12: $\nu^k = 1.1$, with 5,000 instances of each application (1,680 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.986 ($\sigma = 0.00707$, min = 0.972)	0.953 ($\sigma = 0.00768$, min = 0.934)
Round-Robin	0.728 ($\sigma = 0.0993$, min = 0.557)	0.704 ($\sigma = 0.0976$, min = 0.538)
LP_samp(ARITH, 1, 1)	0.983 ($\sigma = 0.0126$, min = 0.935)	0.95 ($\sigma = 0.0143$, min = 0.899)
LP_samp(ARITH, 2, 1)	0.856 ($\sigma = 0.0362$, min = 0.742)	0.827 ($\sigma = 0.0357$, min = 0.723)
LP_samp(GEOM, 2, 1)	0.975 ($\sigma = 0.0165$, min = 0.917)	0.942 ($\sigma = 0.0172$, min = 0.882)
LP_samp(REC, 2, 1)	0.885 ($\sigma = 0.0309$, min = 0.803)	0.855 ($\sigma = 0.0306$, min = 0.775)
LP_samp(ARITH, 2, 2)	0.847 ($\sigma = 0.0641$, min = 0.65)	0.819 ($\sigma = 0.0639$, min = 0.624)
LP_samp(GEOM, 2, 2)	0.948 ($\sigma = 0.0327$, min = 0.78)	0.916 ($\sigma = 0.0331$, min = 0.754)
LP_samp(REC, 2, 2)	0.867 ($\sigma = 0.061$, min = 0.669)	0.838 ($\sigma = 0.0612$, min = 0.64)
LP_samp(ARITH, 4, 1)	0.864 ($\sigma = 0.0541$, min = 0.643)	0.835 ($\sigma = 0.0534$, min = 0.609)
LP_samp(GEOM, 4, 1)	0.955 ($\sigma = 0.0293$, min = 0.776)	0.923 ($\sigma = 0.0297$, min = 0.75)
LP_samp(REC, 4, 1)	0.89 ($\sigma = 0.0441$, min = 0.721)	0.86 ($\sigma = 0.0436$, min = 0.703)
LP_samp(ARITH, 4, 4)	0.87 ($\sigma = 0.0628$, min = 0.672)	0.841 ($\sigma = 0.0633$, min = 0.638)
LP_samp(GEOM, 4, 4)	0.9 ($\sigma = 0.0617$, min = 0.715)	0.87 ($\sigma = 0.0612$, min = 0.677)
LP_samp(REC, 4, 4)	0.869 ($\sigma = 0.0714$, min = 0.583)	0.84 ($\sigma = 0.0706$, min = 0.567)
LP_samp(ARITH, 8, 1)	0.888 ($\sigma = 0.0617$, min = 0.658)	0.858 ($\sigma = 0.0611$, min = 0.625)
LP_samp(GEOM, 8, 1)	0.926 ($\sigma = 0.0419$, min = 0.795)	0.895 ($\sigma = 0.0415$, min = 0.763)
LP_samp(REC, 8, 1)	0.888 ($\sigma = 0.0574$, min = 0.69)	0.858 ($\sigma = 0.0573$, min = 0.655)
0.05-approx	0.995 ($\sigma = 0.0221$, min = 0.862)	0.963 ($\sigma = 0.025$, min = 0.817)
0.2-approx	0.992 ($\sigma = 0.00587$, min = 0.975)	0.958 ($\sigma = 0.00781$, min = 0.934)

Table 5.13: $\nu^k = 10.7$, with 5,000 instances of each application (240 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.856 ($\sigma = 0.104$, min = 0.638)	0.749 ($\sigma = 0.104$, min = 0.556)
Round-Robin	0.773 ($\sigma = 0.115$, min = 0.517)	0.676 ($\sigma = 0.115$, min = 0.45)
LP_samp(ARITH, 1, 1)	0.976 ($\sigma = 0.0294$, min = 0.83)	0.854 ($\sigma = 0.0294$, min = 0.723)
LP_samp(ARITH, 2, 1)	0.888 ($\sigma = 0.0606$, min = 0.691)	0.776 ($\sigma = 0.0606$, min = 0.605)
LP_samp(GEOM, 2, 1)	0.909 ($\sigma = 0.0549$, min = 0.734)	0.795 ($\sigma = 0.0549$, min = 0.643)
LP_samp(REC, 2, 1)	0.903 ($\sigma = 0.0551$, min = 0.716)	0.79 ($\sigma = 0.0551$, min = 0.624)
LP_samp(ARITH, 2, 2)	0.907 ($\sigma = 0.0513$, min = 0.626)	0.793 ($\sigma = 0.0513$, min = 0.545)
LP_samp(GEOM, 2, 2)	0.927 ($\sigma = 0.0413$, min = 0.69)	0.811 ($\sigma = 0.0413$, min = 0.602)
LP_samp(REC, 2, 2)	0.921 ($\sigma = 0.0449$, min = 0.124)	0.805 ($\sigma = 0.0449$, min = 0.108)
LP_samp(ARITH, 4, 1)	0.862 ($\sigma = 0.0781$, min = 0.475)	0.753 ($\sigma = 0.0781$, min = 0.413)
LP_samp(GEOM, 4, 1)	0.864 ($\sigma = 0.079$, min = 0.608)	0.756 ($\sigma = 0.079$, min = 0.529)
LP_samp(REC, 4, 1)	0.859 ($\sigma = 0.0763$, min = 0.586)	0.751 ($\sigma = 0.0763$, min = 0.507)
LP_samp(ARITH, 4, 4)	0.904 ($\sigma = 0.0543$, min = 0.511)	0.791 ($\sigma = 0.0543$, min = 0.445)
LP_samp(GEOM, 4, 4)	0.907 ($\sigma = 0.0525$, min = 0.598)	0.794 ($\sigma = 0.0525$, min = 0.523)
LP_samp(REC, 4, 4)	0.902 ($\sigma = 0.056$, min = 0.354)	0.789 ($\sigma = 0.056$, min = 0.31)
LP_samp(ARITH, 8, 1)	0.837 ($\sigma = 0.0881$, min = 0.436)	0.732 ($\sigma = 0.0881$, min = 0.381)
LP_samp(GEOM, 8, 1)	0.839 ($\sigma = 0.0885$, min = 0.504)	0.734 ($\sigma = 0.0885$, min = 0.436)
LP_samp(REC, 8, 1)	0.83 ($\sigma = 0.0899$, min = 0.476)	0.726 ($\sigma = 0.0899$, min = 0.415)
0.05-approx	0.999 ($\sigma = 0.00737$, min = 0.807)	0.874 ($\sigma = 0.00737$, min = 0.708)
0.2-approx	0.985 ($\sigma = 0.00897$, min = 0.929)	0.861 ($\sigma = 0.00897$, min = 0.809)

Table 5.14: $\phi = 0$, with 5,000 instances of each application (3,040 simulations).

Correlation factor. Tables 5.14, 5.15, and 5.16 show that On-Demand, Round-Robin, and our ε-approximation are mostly independent of the correlation between computations and communications. On the contrary, the relative performance of LP_samp(m, c, 1) is increased when there is a correlation between both values ($\phi > 0$). However, these heuristics remain worse than LP_samp(m, 1, 1), using a single bucket for each application.

5.5 Conclusion and perspectives

In this chapter, we have focused our attention on the scheduling of multiple bags of tasks on a star-shaped heterogeneous platform. We considered the case of tasks, whose characteristics follow probability laws, rather than being constant. We presented an ε-approximation using a rational linear program, appropriate for the *off-line* model, as well as a heuristic suitable for the *online* model. We compared these method to two well-known policies, Round-Robin and On-Demand, with a large number of simulations. These simulations essentially showed that the best method is our *varepsilon*-approximation, which is an off-line algorithm, and our LP_samp(1) heuristic, which a semi-clairvoyant method. Round-Robin is outperformed by other methods, but also that On-Demand performed very well on average, but offers a throughput up to 30% smaller than our steady-state schedules. We only used the first instances of each application to get a sufficient knowledge before running LP_samp(1) once. We could improve this method by running LP_samp(1) every n instances, using our knowledge about these n instances to obtain a better representation of the distribution.

In this chapter, we used a related heterogeneous model for computations, but the linear programming approach we used could be easily extended to an unrelated model. In comparison, dynamic policies like On-Demand do not take into account this difference and should perform worse.

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.863 (σ = 0.0979, min = 0.658)	0.756 (σ = 0.0979, min = 0.577)
Round-Robin	0.779 (σ = 0.111, min = 0.509)	0.683 (σ = 0.111, min = 0.445)
LP_samp(ARITH, 1, 1)	0.982 (σ = 0.0195, min = 0.898)	0.86 (σ = 0.0195, min = 0.784)
LP_samp(ARITH, 2, 1)	0.912 (σ = 0.0441, min = 0.732)	0.799 (σ = 0.0441, min = 0.64)
LP_samp(GEOM, 2, 1)	0.937 (σ = 0.0332, min = 0.789)	0.821 (σ = 0.0332, min = 0.693)
LP_samp(REC, 2, 1)	0.93 (σ = 0.0341, min = 0.788)	0.815 (σ = 0.0341, min = 0.688)
LP_samp(ARITH, 2, 2)	0.913 (σ = 0.0494, min = 0.65)	0.8 (σ = 0.0494, min = 0.561)
LP_samp(GEOM, 2, 2)	0.925 (σ = 0.0406, min = 0.61)	0.81 (σ = 0.0406, min = 0.533)
LP_samp(REC, 2, 2)	0.918 (σ = 0.0428, min = 0.669)	0.805 (σ = 0.0428, min = 0.576)
LP_samp(ARITH, 4, 1)	0.895 (σ = 0.0539, min = 0.573)	0.784 (σ = 0.0539, min = 0.499)
LP_samp(GEOM, 4, 1)	0.9 (σ = 0.05, min = 0.674)	0.789 (σ = 0.05, min = 0.587)
LP_samp(REC, 4, 1)	0.894 (σ = 0.0496, min = 0.684)	0.783 (σ = 0.0496, min = 0.598)
LP_samp(ARITH, 4, 4)	0.906 (σ = 0.0515, min = 0.528)	0.794 (σ = 0.0515, min = 0.464)
LP_samp(GEOM, 4, 4)	0.908 (σ = 0.0502, min = 0.687)	0.796 (σ = 0.0502, min = 0.599)
LP_samp(REC, 4, 4)	0.901 (σ = 0.0563, min = 0.642)	0.789 (σ = 0.0563, min = 0.562)
LP_samp(ARITH, 8, 1)	0.877 (σ = 0.0604, min = 0.617)	0.768 (σ = 0.0604, min = 0.539)
LP_samp(GEOM, 8, 1)	0.881 (σ = 0.0586, min = 0.626)	0.772 (σ = 0.0586, min = 0.547)
LP_samp(REC, 8, 1)	0.872 (σ = 0.0617, min = 0.571)	0.764 (σ = 0.0617, min = 0.497)
0.05-approx	0.999 (σ = 0.00345, min = 0.944)	0.876 (σ = 0.00345, min = 0.813)
0.2-approx	0.985 (σ = 0.0106, min = 0.709)	0.863 (σ = 0.0106, min = 0.621)

Table 5.15: $\phi = 0.5$, with 5,000 instances of each application (3,040 simulations).

Heuristic	Normalized to best	Normalized to UB
On-Demand	0.871 (σ = 0.0919, min = 0.677)	0.763 (σ = 0.0919, min = 0.595)
Round-Robin	0.784 (σ = 0.111, min = 0.541)	0.688 (σ = 0.111, min = 0.472)
LP_samp(ARITH, 1, 1)	0.995 (σ = 0.0058, min = 0.954)	0.872 (σ = 0.0058, min = 0.834)
LP_samp(ARITH, 2, 1)	0.933 (σ = 0.0389, min = 0.742)	0.818 (σ = 0.0389, min = 0.651)
LP_samp(GEOM, 2, 1)	0.959 (σ = 0.026, min = 0.803)	0.84 (σ = 0.026, min = 0.702)
LP_samp(REC, 2, 1)	0.952 (σ = 0.0281, min = 0.812)	0.834 (σ = 0.0281, min = 0.709)
LP_samp(ARITH, 2, 2)	0.924 (σ = 0.0461, min = 0.675)	0.81 (σ = 0.0461, min = 0.59)
LP_samp(GEOM, 2, 2)	0.932 (σ = 0.0404, min = 0.701)	0.817 (σ = 0.0404, min = 0.611)
LP_samp(REC, 2, 2)	0.927 (σ = 0.0389, min = 0.721)	0.813 (σ = 0.0389, min = 0.632)
LP_samp(ARITH, 4, 1)	0.926 (σ = 0.0434, min = 0.688)	0.812 (σ = 0.0434, min = 0.601)
LP_samp(GEOM, 4, 1)	0.933 (σ = 0.0383, min = 0.734)	0.817 (σ = 0.0383, min = 0.643)
LP_samp(REC, 4, 1)	0.927 (σ = 0.0389, min = 0.721)	0.813 (σ = 0.0389, min = 0.632)
LP_samp(ARITH, 4, 4)	0.914 (σ = 0.0502, min = 0.684)	0.801 (σ = 0.0502, min = 0.596)
LP_samp(GEOM, 4, 4)	0.915 (σ = 0.0493, min = 0.574)	0.802 (σ = 0.0493, min = 0.503)
LP_samp(REC, 4, 4)	0.905 (σ = 0.0564, min = 0.568)	0.794 (σ = 0.0564, min = 0.499)
LP_samp(ARITH, 8, 1)	0.917 (σ = 0.047, min = 0.701)	0.804 (σ = 0.047, min = 0.611)
LP_samp(GEOM, 8, 1)	0.921 (σ = 0.0445, min = 0.672)	0.807 (σ = 0.0445, min = 0.588)
LP_samp(REC, 8, 1)	0.914 (σ = 0.0488, min = 0.674)	0.801 (σ = 0.0488, min = 0.588)
0.05-approx	0.999 (σ = 0.00291, min = 0.986)	0.876 (σ = 0.00291, min = 0.859)
0.2-approx	0.988 (σ = 0.00913, min = 0.962)	0.866 (σ = 0.00913, min = 0.842)

Table 5.16: $\phi = 1$, with 5,000 instances of each application (3,040 simulations).

Chapter 6

Computing the throughput of replicated workflows

6.1 Introduction

In this chapter, we focus our attention on simplified task graphs, as we only work on streaming applications, or *workflows*, whose dependency graph is a linear chain composed of several tasks, or *stages*. Such applications operate on a collection of data sets that are executed in a pipeline fashion [75, 74, 81]. They are a popular programming paradigm for streaming applications like video and audio encoding and decoding, DSP applications, etc. [39, 78, 86].

We still consider a large number of instances of the same workflow executed on a large heterogeneous computing platform. As explained in Chapter 3, we aim at maximizing the throughput ρ of our platform in steady-state, or, equivalently, at minimizing its period $\mathcal{T}$, which is defined as the inverse of the throughput.

When mapping application tasks onto processors, we enforce the rule that any given processor will execute at most one task. However, the converse is not true. If the computations of a given task are independent from one data set to another, then two consecutive computations (different data sets) for the same task can be mapped onto distinct processors. Such a task is said to be *replicated*, using the terminology of Subhlok and Vondran [75, 76] and of the DataCutter team [26, 74, 82]. This corresponds to the *dealable stages* of Cole [35]. Thus, contrarily to Chapter 4, we deal in this chapter with multi-allocations mappings.

Given an application and a target heterogeneous platform, the problem to determine the optimal mapping (maximizing the throughput) has been shown NP-hard in [23]. The main objective of this chapter is to assess the complexity of computing the throughput *when the mapping and the order of communications are given*. The problem is easy when workflow tasks are not replicated, i.e., when each task is assigned to a single processor: in that case the period is dictated by the critical hardware resource. But when tasks are replicated, i.e., when a task is assigned to several processors, the problem gets surprisingly complicated, and we provide examples where the optimal period is larger than the largest cycle-time of any resource. In other words, during the execution of the system, any resource will be idle at some point. We present in Section 6.2 the detailed framework. We then show a model based on timed Petri nets in Section 6.3 and how to use them in Section 6.4 to compute the optimal period in the general case, and we provide a polynomial algorithm for the one-port model with overlap. Finally, we report in Section 6.5 comprehensive simulation results on the gap between the optimal period and the largest resource cycle-time, before concluding this chapter in Section 6.6.

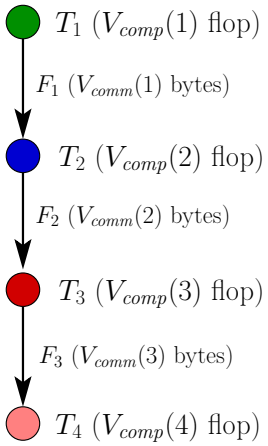


Figure 6.1: Example of a 4-stage pipeline.

6.2 Notations and hypotheses

Most notations were already defined in Section 3.2, we only briefly recall these definitions and complete them by notations that are specific to this chapter.

6.2.1 Application model

As said in the introduction, we restrain our applications to streaming ones, or *workflows*, whose dependency graph is a linear chain composed of m tasks, called T_k ($1 \leq k \leq m$). Each task T_k has a size $V_{comp}(k)$, expressed in FLOP. Compared to general applications presented in Section 3.2, the dependency scheme is simplified: T_k needs an input file F_{k-1} of size $V_{comm}(k-1)$, expressed in BYTES and produces an output file F_k of size $V_{comm}(k)$, which is the input file of stage T_{k+1}. All these sizes are independent of the data set. Note that T_1 produces the initial data and does not receive any input file ($V_{comm}(0) = 0$), while T_m gathers the final data and does not send any file ($V_{comm}(m) = 0$). Figure 6.1 shows a simple example of a 4-task pipeline.

6.2.2 Platform model

The workflow is executed on a fully connected, heterogeneous platform G_P with n processors; the timed needed by P_u to process T_k is equal to $w_{u,k}$, following the unrelated model. We assume that $P_u \rightarrow P_v$ is a bidirectional link from P_u to P_v, with bandwidth $bw_{u,v}$. For instance, we can have a physical star-shaped platform, where all processors are linked to each other through a central switch. The time needed to transfer a file F_i from P_u to P_v is $\frac{V_{comm}(i)}{bw_{u,v}}$. Two realistic common models are used for communications:

Overlap One-Port. This first model permits overlap of communications by computations: any processor can simultaneously receive data set $i+1$, compute the result of data set i, and send the resulting data set $i-1$ to the next processor. Requiring multi-threaded programs and full-duplex network interfaces, this model allows for a better use of computational resources.

Strict One-Port. In this model, there is no overlap of communications by computations: a processor can either receive a given set of data, compute its result, or send this result. This is the typical execution of a single-threaded program, with one-port serialized communications. Although leading to a less efficient use of physical resources, this model allows for simpler programs and hardware.

6.2.3 Replication model

If we assume that a processor can process at most a single application task, any task can be replicated onto several processors. Since instances of a same task are independent from one data set to another, two successive computations are done onto distinct processors without extra communications. Note that the computations of a replicated task can be fully sequential for a given data set, what matters is that they do not depend from previous results for other data sets, hence the possibility to process different data sets in different locations. The following schema illustrates the replication of a task T_k onto three processors:

$$\ldots\ T_{k-1} \begin{cases} \nearrow & T_k \text{ on } P_1\text{: data sets } \mathbf{1, 4, 7, \ldots} \\ - & T_k \text{ on } P_2\text{: data sets } \mathbf{2, 5, 8, \ldots} \\ \searrow & T_k \text{ on } P_3\text{: data sets } \mathbf{3, 6, 9, \ldots} \end{cases} \begin{cases} \searrow \\ - \\ \nearrow \end{cases} T_{k+1}\ \ldots$$

We remark that if no other task is replicated, then we have three distinct allocations. Since the platform is said to be a fully-connected graph, any allocation is completely defined by giving $\sigma(T_k)$ for all tasks.

To fully describe a schedule, we obviously need the mapping of all tasks to processors but we also need the order of processors for the Round-Robin distribution of instances on processors. If task T_k is replicated onto a tuple (P_3, P_4, P_7), then we assume that the first instance is processed by P_3, the second one is processed by P_4, the third one by P_7, the fourth one by P_3, and so on. This order is required to determine which communication links are used, and is respected even if another processor is available. In the following pages, we assume without any loss of generality that if we have $i < j$, then P_i processes its first instance before P_j. By definition, all processors given in a tuple must be distinct.

The objective is to maximize the throughput ρ of the system. Equivalently, we aim at minimizing the period $\mathcal{T}$, which is the inverse of the throughput and corresponds to the time-interval that separates two consecutive data sets entering the system. We can derive a lower bound for the period as follows. Let $C_{\text{exec}}(i)$ be the cycle-time of processor P_i. If we enforce the Overlap One-Port model, then $C_{\text{exec}}(i)$ is equal to the maximum of its reception time $C_{\text{in}}(i)$, its computation time $C_{\text{comp}}(i)$, and its transmission time $C_{\text{out}}(i)$ (assuming that $C_{\text{in}}(1) = C_{\text{out}}(n) = 0$):

$$C_{\text{exec}}(i) = \max\left\{C_{\text{in}}(i), C_{\text{comp}}(i), C_{\text{out}}(i)\right\}.$$

If we enforce the Strict One-Port model, then $C_{\text{exec}}(i)$ is equal to the sum of the three operations:

$$C_{\text{exec}}(i) = C_{\text{in}}(i) + C_{\text{comp}}(i) + C_{\text{out}}(i).$$

In both models, the maximum cycle-time, $\mathcal{M}_{\text{ct}} = \max_{1 \leq i \leq n} C_{\text{exec}}(i)$, is a lower bound for the period.

Given an application and a target heterogeneous platform, determining a mapping which maximizes the throughput has been shown to be an NP-hard problem in [23], even in the simple case where no task can be replicated (thereby enforcing a one-to-one mapping of tasks

to processors). The proof of [23] was given for the STRICT ONE-PORT model but can be easily extended to the OVERLAP ONE-PORT model. In this chapter, we deal with the following problem, which in appearance looks simpler: given the mapping of tasks to processors, how can we compute the period $\mathcal{T}$? If no task is replicated, then the period is simply determined by the critical resource (maximum cycle-time): $\mathcal{T} = \mathcal{M}_{\text{ct}}$. Again, this problem is addressed in [23] for the STRICT ONE-PORT model but the same result can be easily shown for the OVERLAP ONE-PORT model. However, when tasks are replicated, the previous result is no longer true, and we need to use more sophisticated techniques such as timed Petri nets.

6.3 Timed Petri net models

6.3.1 A short introduction to timed Petri nets

In this section, we aim at modeling mappings with timed Petri nets (TPNs) as defined in [10], in order to be able to compute the period of a given mapping. In the following, only TPNs with the *event graph property* will be considered (see [11]). First, we recall the fundamentals of Petri nets.

A timed Petri net is a 5-tuple (S, T, W, ν, M_0), where:

- S is a finite set of places,
- T is a finite set of transitions, such that $T \cap S = \emptyset$,
- $W : (S \times T) \cup (T \times S) \to \mathbb{N}$ defines a set of weighted arcs from a transition $t \in T$ to a place $p \in S$, or from a place p to a transition t: if $W(x, y)$ is a positive integer, then there is an arc of weight $W(x, y)$ from x to y. Otherwise, $W(x, y)$ is equal to 0.
- $\nu : T \to \mathbb{R}_+$ specifies the time required to fire a transition.
- $M_0 : S \to \mathbb{N}$ is the initial marking, i.e., a function associating an initial number of tokens to places.

A place p is an *input* place of a transition t if, and only if, there is an arc from p to t: $W(p, t) > 0$. Similarly, p is an output place of t if, and only if, there is an arc from t to p: $W(t, p) > 0$.

A transition t is enabled if any input place p of t has enough tokens: $M_0(p) \geq W(p, t)$. If t is *fired* at time τ, then the content of places connected to p are changed: tokens of input places are consumed at time τ and some tokens are created on output places $\tau + \nu(t)$. The new marking M is defined at time τ by:

$$\forall p \in S, M(p) = M_0(p) - W(p, t).$$

At time $\tau + \nu(t)$, tokens are created and the definitive marking M' is defined by:

$$\forall p \in S, M'(p) = M(p) + W(t, p).$$

As said before, we only consider *event graphs*: each place of the Petri net has exactly one input and one output transition.

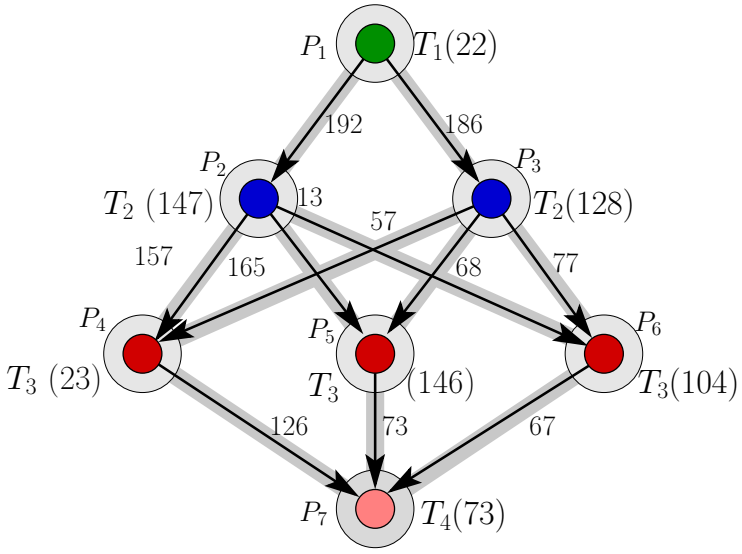


Figure 6.2: Example A: Mapping with replication: T_2 on 2 processors, T_3 on 3 processors.

6.3.2 Mappings with replication

We consider mappings where some tasks may be replicated, as defined in Section 6.2.3: a task can be processed by one or more processors. As already stated, two rules are enforced to simplify the model: a processor can process at most one task, and if several processors are involved in the computation of one task, they are served in a Round-Robin fashion. For example, if T_k is mapped onto the tuple (P_1, P_2), P_1 processes each odd instance of T_k starting from the first one and P_2 processes each even instance, even if P_1 is faster than P_2. In all our Petri net models, the use of a physical resource during a time t (i.e., the computation of a task 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 tasks replicated on different processors. Consider Example A described in Figure 6.2: the first data set enters the system and proceeds through processors P_1, P_2, P_4 and P_7. The second data set is first processed by processor P_1, then by processor P_3 (even if P_2 is available), by processor P_5 and finally by processor P_7. Paths followed by the first eight input data sets are summarized up in Table 6.1: 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 $i - 6$. Since the platform is a fully-connected graph, the path followed by a data set is equivalent to an allocation. We have the following easy result:

Proposition 6.1. *Consider a pipeline of m tasks $T_1, \ldots, T_m$, such that task T_i is mapped onto R_i distinct processors. Then the number of paths followed by the input data in the whole system, or, equivalently, the number of allocations, is equal to $R = \mathrm{lcm}\,(R_1, \ldots, R_m)$.*

Proof. Let R be the number of paths $\mathcal{P}_j$ followed by the input data. Assume that task T_i is processed by processors $P_{i,0}, \ldots, P_{i,R_i-1}$. By definition, all paths are distinct. Moreover, the Round-Robin order is respected: path $\mathcal{P}_j$ is made of processors $(P_{1,j \bmod R_1}, \ldots, P_{i,j \bmod R_i}, \ldots, P_{m,j \bmod R_m})$. The first path $\mathcal{P}_0$ is made of $(P_{1,0}, P_{2,0}, \ldots, P_{m,0})$. By definition, R is the

Input data	Path in the system
0	$P_1 \to P_2 \to P_4 \to P_7$
1	$P_1 \to P_3 \to P_5 \to P_7$
2	$P_1 \to P_2 \to P_6 \to P_7$
3	$P_1 \to P_3 \to P_4 \to P_7$
4	$P_1 \to P_2 \to P_5 \to P_7$
5	$P_1 \to P_3 \to P_6 \to P_7$
6	$P_1 \to P_2 \to P_4 \to P_7$
7	$P_1 \to P_3 \to P_5 \to P_7$

Table 6.1: Example A: Paths followed by the first input data.

smallest positive integer, such that the $(R+1)$-th used path is identical to the first one:

$$\forall i \in \{1, \ldots, m\}, R \mod R_i = 0.$$

R is then the smallest positive integer that is divisible by each R_i, i.e., $R = \text{lcm}(R_1, \ldots, R_m)$. ■

The TPN model described in this chapter is rather similar to what has been done to model jobshops with static schedules using TPNs [53]. Here, however, replication imposes that each path followed by the input data must be fully developed in the TPN: if P_1 appears in several distinct paths, as in Figure 6.2, there are several transitions corresponding to P_1. 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.

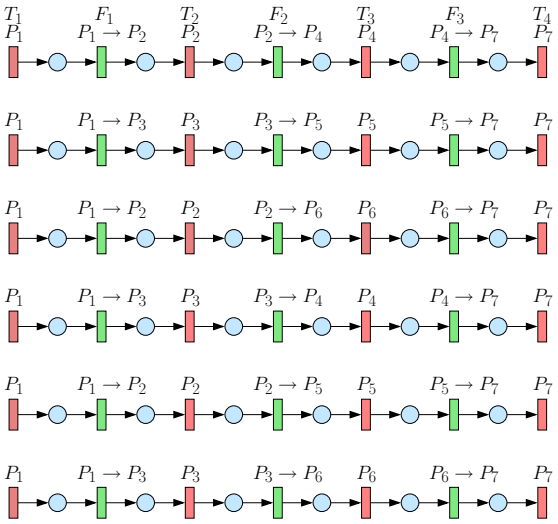
6.3.3 Overlap One-Port model

First, let us focus on the Overlap One-Port model: any processor can receive a file and send another one while computing. All allocations or paths followed by the input data in the whole system have to appear in the TPN. We use the notations of Proposition 6.1.

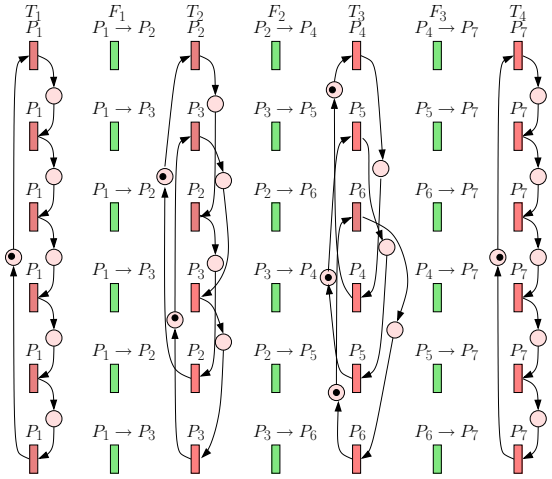
Let R denote the number of allocations of our mapping. Then the i-th input data follows the $(i \mod R)$-th allocation, and we have a rectangular TPN, with R rows of $2m-1$ transitions, due to the m transitions representing the use of processors and the $m-1$ transitions representing the use of communication links. The i-th transition of the j-th row is named Tr_i^j. The time required to fire a transition Tr_{2i}^j (corresponding to the processing of task T_i on processor P_u) is set to $w_{u,k}$, and the one required by a transition Tr_{2i+1}^j (corresponding to the transmission of a file F_i from P_u to P_v) is set to $V_{comm}(i)/bw_{u,v}$.

Then we add places between these transitions to model the following set of constraints:

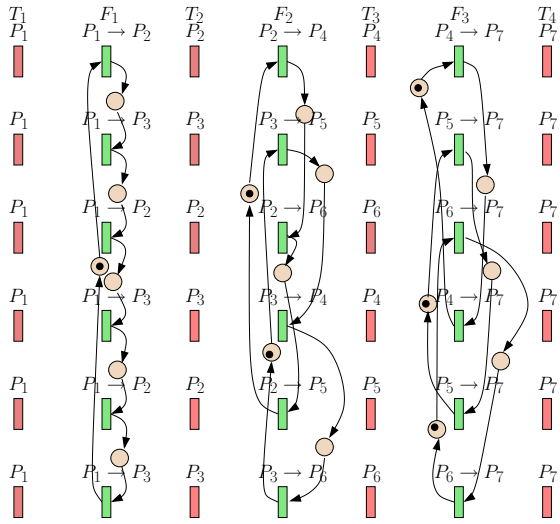
1. The file F_i cannot be sent before the computation of T_i : a place is added from Tr_{2i}^j to Tr_{2i+1}^j on each row. Similarly, the task T_{i+1} cannot be processed before the end of the communication of F_i: a place is added from Tr_{2i+1}^j to $Tr_{2(i+1)}^j$ on each row j. All these places are shown in Figure 6.3(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 $Tr_{2i}^{j_l}$ to $Tr_{2i}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $Tr_{2i}^{j_k}$ to $Tr_{2i}^{j_1}$. All these places are shown in Figure 6.3(b).



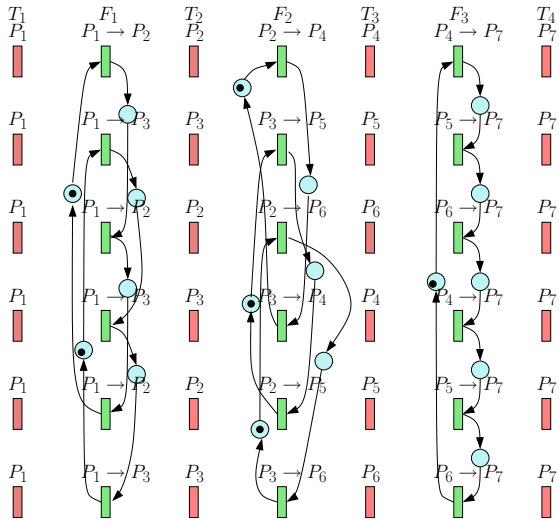
(a) Dependencies between communications and computations.



(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 6.3: Overlap One-Port model: places imposed by the different constraints described in Subsection 6.3.3. 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.

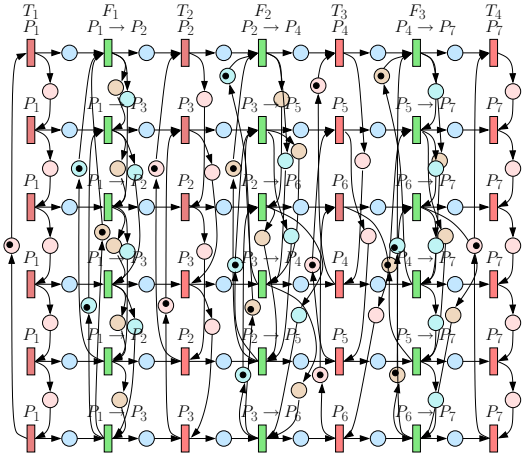


Figure 6.4: Complete TPN of Example A for the Overlap One-Port model.

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 $j_1, j_2, \ldots, j_k$ and does not compute the last task. Then we add a place from $Tr_{2i+1}^{j_l}$ to $Tr_{2i+1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $Tr_{2i+1}^{j_k}$ to $Tr_{2i+1}^{j_1}$ to ensure that P_i does not send two files simultaneously. All these places are shown in Figure 6.3(c).
4. In the same way, if P_i does not compute the first task, we add a place from $Tr_{2i-1}^{j_l}$ to $Tr_{2i-1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $Tr_{2i-1}^{j_k}$ to $Tr_{2i-1}^{j_1}$ to ensure that P_i does not receive two files simultaneously. All these places are shown in Figure 6.3(d).

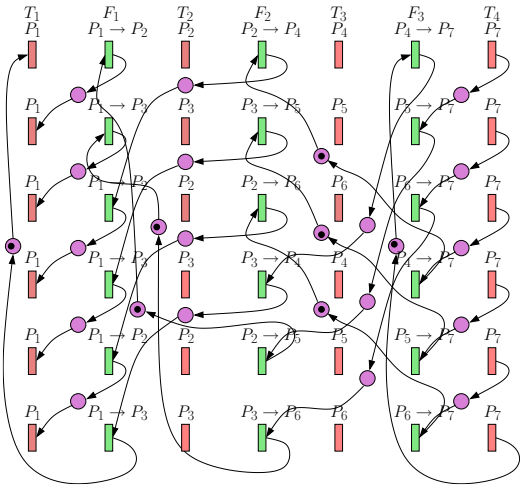
Finally, any resource, before it is use for the first time, is ready to compute or communicate, and is waiting only for the input file. Indeed, a token is put in every place going from a transition $Tr_i^{j_k}$ to a transition $Tr_i^{j_1}$, as defined in the previous lines. The complete TPN of Example A for the Overlap One-Port model is given in Figure 6.4.

6.3.4 Strict One-Port model

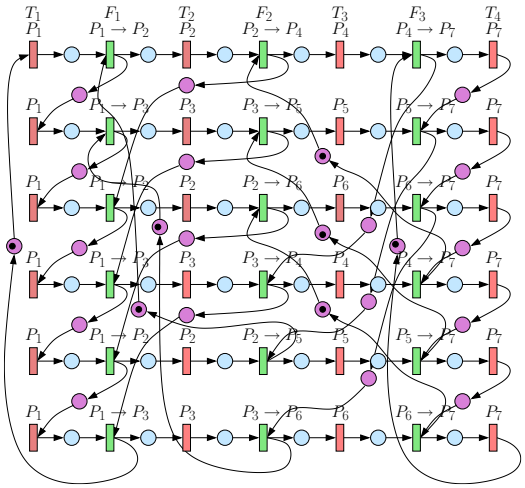
In the Strict One-Port model, any processor can either send a file, receive another one, or perform a computation while these operations were happening concurrently in the Overlap One-Port model. Hence, we require a processor to successively receive the data corresponding to an input file F_i, compute the task T_{i+1} and send the file F_{i+1} before receiving the next data set of F_i. Allocations are obviously the same as in Subsection 6.3.3, and the structure of the TPN remains the same (R rows of $2m-1$ transitions).

The first set of constraints is also identical to that of the Overlap One-Port model, since we still have dependencies between communications and computations, as in Figure 6.3(a). However, the other dependencies are replaced by those imposed by the Round-Robin order of the Strict One-Port model.

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 $Tr_{2i+1}^{j_l}$ to $Tr_{2i-1}^{j_{l+1}}$ with $1 \leq l \leq k-1$, and a place from $Tr_{2i+1}^{j_k}$ to $Tr_{2i-1}^{j_1}$. These places ensure 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 6.5(a).



(a) Dependencies imposed by the Round-Robin distribution.



(b) Complete TPN.

Figure 6.5: STRICT ONE-PORT model: places imposed by the different constraints described in Subsection 6.3.4.

Any physical resource can immediately start its first communication, since it is initially waiting only for the input file. Thus a token is put in every place from a transition $Tr_i^{j_k}$ to a transition $Tr_i^{j_1}$, as defined in the previous lines. The complete TPN of Example A for the STRICT ONE-PORT model is given in Figure 6.5(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: $\mathcal{O}(mn)$.

6.4 Computing mapping throughputs

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 [11]. For any cycle $\mathcal{C}$ in the TPN, let $\mathcal{L}(\mathcal{C})$ be its length (the sum of the time of its 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 $\max_{\mathcal{C}\text{cycle}} \mathcal{L}(\mathcal{C})/t(\mathcal{C})$, and this ratio is the period $\mathcal{T}$ of the system: indeed, after a transitive period, every transition of the TPN is fired exactly once during a period of length $\mathcal{T}$ [11].

Critical cycles can be computed with softwares like ERS [57] or GreatSPN [34] with a complexity $\mathcal{O}(R^3m^3)$. By definition of the TPN, the firing of any transition of the last column corresponds to the completion of the last task, i.e., to the completion of an instance of the workflow. Moreover, we know that all the R transitions (if R is still the number of rows of the TPN) of this last column are fired in a Round-Robin order. In our case, R data sets are completed during any period $\mathcal{T}$: the obtained throughput ρ is $\frac{R}{\mathcal{T}}$.

6.4.1 Overlap One-Port model

The TPN associated to the OVERLAP ONE-PORT model 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, any cycle only contains

transitions belonging to the same "column": we can split the complete TPN into $2m-1$ 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 $R = \text{lcm}\,(R_1, \ldots, R_m)$.

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. This is the main technical contribution of this chapter, given in the following Theorem 6.1.

Before explaining this theorem and proving it, we present a result, giving the number of connected components of the sub-TPN corresponding to a single communication. If a given processor P_i sends data to P_{j_1} and to P_{j_2} (or receive data from them), then there is a dependency between the P_{j_1} and P_{j_2}. In the TPN, this dependency is expressed by a place between the transition corresponding to the communication from P_i to P_{j_1} and the transition corresponding to the communication from P_i to P_{j_2}. More generally, if we consider the dependency graph, then two transitions belong to the same connected component if, and only if, there is a set of dependencies from one to the other. Due to the Round-Robin distribution, being in the connected component is an equivalence relation.

Lemma 6.1. *Assume that the communication of any file F_i involves a senders and b receivers. The graph of dependencies between transitions is made of* $\gcd(a,b)$ *connected components of the same size.*

Proof. Consider any communication between a senders and b receivers. Without any loss of generality, assume that senders are named $P_0, P_1, \ldots, P_{a-1}$, and that receivers are named $Q_0, Q_1, \ldots, Q_{b-1}$. Let P_{i_k} be the sender of the transmission of the k-th instance, while Q_{j_k} is the receiver. By definition of the Round-Robin distribution, we have:

$$i_k \equiv k \pmod{a} \quad \text{and} \quad j_k \equiv k \pmod{b}. \tag{6.1}$$

A dependency exists between the k-th instance and the k'-th one if, and only if, they have the same receiver ($P_{i_k} = P_{i_{k'}}$) or the same sender ($Q_{j_k} = Q_{j_{k'}}$). Thanks to Equation (6.1), there exist two integers x and y, verifying:

$$k' = k + ax \quad \text{and} \quad k' = k + by.$$

Reciprocally, if there exist two integers x and y, such that $k' = k + ax + by$, then there is a dependency between the transitions corresponding to the k-th instance and to the k'-th one.

Thus, the transitions corresponding to the k-th instance and to the k'-th one are in the same connected component if, and only if, there exist x and y, such that:

$$k - k' = ax + by.$$

This is a classical Diophantian equation, which has solutions if, and only if, $k - k'$ is a multiple of the greatest common divisor d of a and b ($d = \gcd(a,b)$).

Thus, we have d equivalence classes, corresponding to d connected components, which are fully determined by the communication of the first d instances. Moreover, the i-th connected component is made of communications between processors P_{i+xd} (with $0 \leq x < a/d$) and processors Q_{j+yd} (with $0 \leq y < b/d$).

■

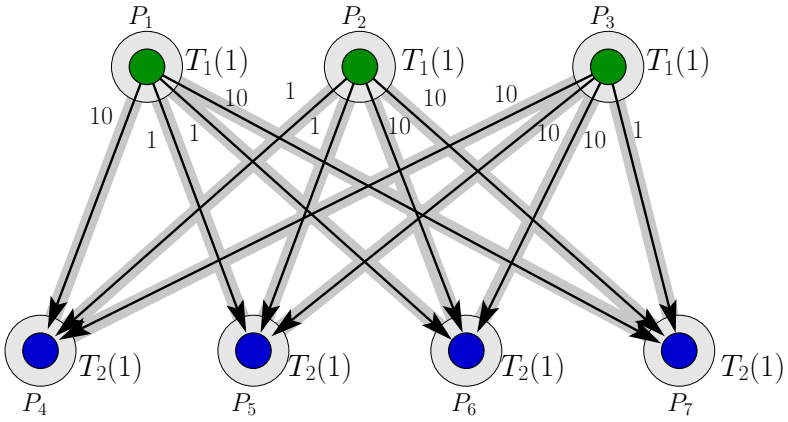


Figure 6.6: Example B: T_1 is replicated on 3 processors, and T_2 on 4 processors.

Theorem 6.1. *Consider a pipeline of m tasks $T_1, \ldots, T_m$, such that task T_i is mapped onto R_i distinct processors. Then the throughput of this system can be computed in time $\mathcal{O}\left(\sum_{i=1}^{m-1}\left((R_i R_{i+1})^3\right)\right)$.*

Proof. We saw that the throughput of the platform is given by the weight of a critical cycle. As said before, a critical cycle can only be found in a column of transitions, and we have two cases:

- transitions correspond to the computation of a task T_i,
- transitions correspond to the transmission of a file F_i.

The first case is the simplest one: each transition appears in exactly one cycle, and each cycle passes through exactly one physical resource (all the transitions correspond to the same task T_i on the same processor P_u). The average time required by P_u to process a single instance is equal to $\left(\frac{w_{u,i}}{R_i}\right)$ (we recall that T_i is replicated onto R_i distinct processors). Thus, the running time to compute cycle times for those columns is $\mathcal{O}\left(\sum_{i=1}^{m} R_i\right)$.

The second case is more complex: each transition appears in exactly two cycles. The first cycle is created by the Round-Robin distribution on the output port of the emitter, and the second one comes from the Round-Robin distribution on the input port of the receiver. By construction, the corresponding sub-TPN is made of several elemental cycles, each elemental cycle corresponding to the successive receptions of F_i by a processor participating to the computation of T_{i+1}, or to the successive transmissions of F_i by a processor working on T_i. If any critical cycle passes through both types of elemental cycles, then all resources can have idle times in the final schedule. This is shown by Figure 6.10, which represents the Gantt chart of the first instances of Example B. This example, presented in Figure 6.6, is made of a single communication, whose sub-TPN is displayed in Figure 6.8; a critical cycle is drawn with dotted arrows.

The communication of F_i involves R_i senders and R_{i+1} receivers. The transmission of F_1 in Example C, displayed in Figure 6.9, is used for a better understanding of our proof. Let R be the least common multiple of $(R_1, \ldots, R_m)$, and p the greatest common divisor of R_i and R_{i+1} ($p = \gcd(R_i, R_{i+1})$). Let u be equal to R_i/p and v be equal to R_{i+1}/p. Thanks to Lemma 6.1 the complete sub-TPN (denoted by $\mathcal{G}$) is made of p connected components. Each of them are based on $c = R/\mathrm{lcm}(R_i, R_{i+1})$ copies of a pattern P of size $u \times v$. If T_i is replicated on $A_0, \ldots, A_{R_i-1}$ and T_{i+1} is replicated on $B_0, \ldots, B_{R_i-1}$, then this pattern P corresponds to communications between processors A_{i+xd} (with $0 \leq x < u$) and processors B_{j+yp} (with $0 \leq y < v$). One of

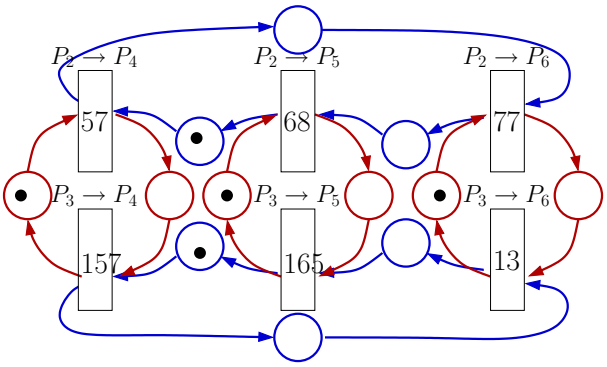


Figure 6.7: Sub-TPN corresponding to the transmission of F_1 in Example A (Overlap One-Port model).

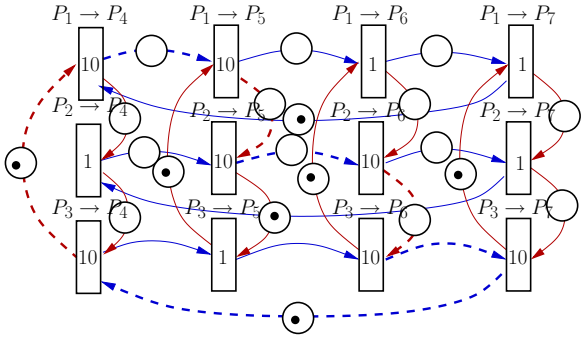


Figure 6.8: Sub-TPN corresponding to the transmission of F_1 in Example B (Overlap One-Port model).

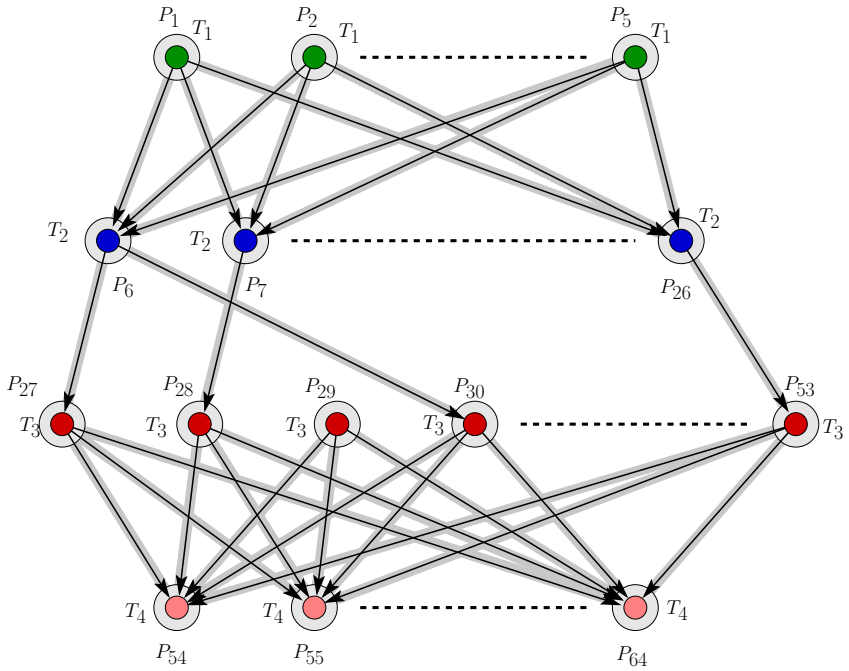


Figure 6.9: Example C: Tasks are respectively replicated on 5, 21, 27 and 11 processors.

these components is shown in Figure 6.11. Let $\mathcal{C}^a$ be a critical cycle. By definition of a cycle, $\mathcal{C}^a$ is contained in one of the p connected components. Thus, without any loss of generality, we now assume that the complete sub-TPN is reduced to a single connected component.

In the case of Example C, $R_1 = 5$, $R_2 = 21$, $R_3 = 27$ and $R_4 = 11$. Thus, we have $R = 10395$, $p = 3$, $c = 55$, $u = 7$ and $v = 9$. There are 3 connected components, reflecting the fact that any sender communicates with only 9 distinct receivers. For example, P_6 only communicates with P_{27}, P_{30}, P_{33}, ..., P_{51}, and P_7 only communicates with P_{28}, P_{31}, P_{34}, ..., P_{52}.

Let us call x_{ij}^k the transition on column i ($0 \leq i < u$), row j ($0 \leq j < v$) and pattern k ($0 \leq k < c$).

The structure of any connected component is very regular:

- if $0 \leq i < u$, then there is a place from x_{ij}^k to $x_{(i+1)j}^k$, corresponding to the Round-Robin on the receiver,
- if $0 \leq j < v$, then there is a place from x_{ij}^k to $x_{i(j+1)}^k$, corresponding to the Round-Robin on the sender,
- if $0 \leq k < c$, then there is a place from $x_{(u-1)j}^k$ to x_{0j}^{k+1} and from $x_{i(v-1)}^k$ to x_{i0}^{k+1},
- there is a place from $x_{(u-1)j}^{c-1}$ to x_{0j}^0 and from $x_{i(v-1)}^{c-1}$ to x_{i0}^0.

Thus, any critical cycle passes through all patterns of $\mathcal{G}$. Now, let us call $\mathcal{G}'$ the smaller graph made of a single pattern of $\mathcal{G}$. $\mathcal{G}'$ has uv transitions, denoted by x_{ij} (with $0 \leq i < u$ and $0 \leq j < vb$) and $2uv$ places, such that:

- if $0 \leq i < u$, then there is a place from x_{ij} to $x_{(i+1)j}$,

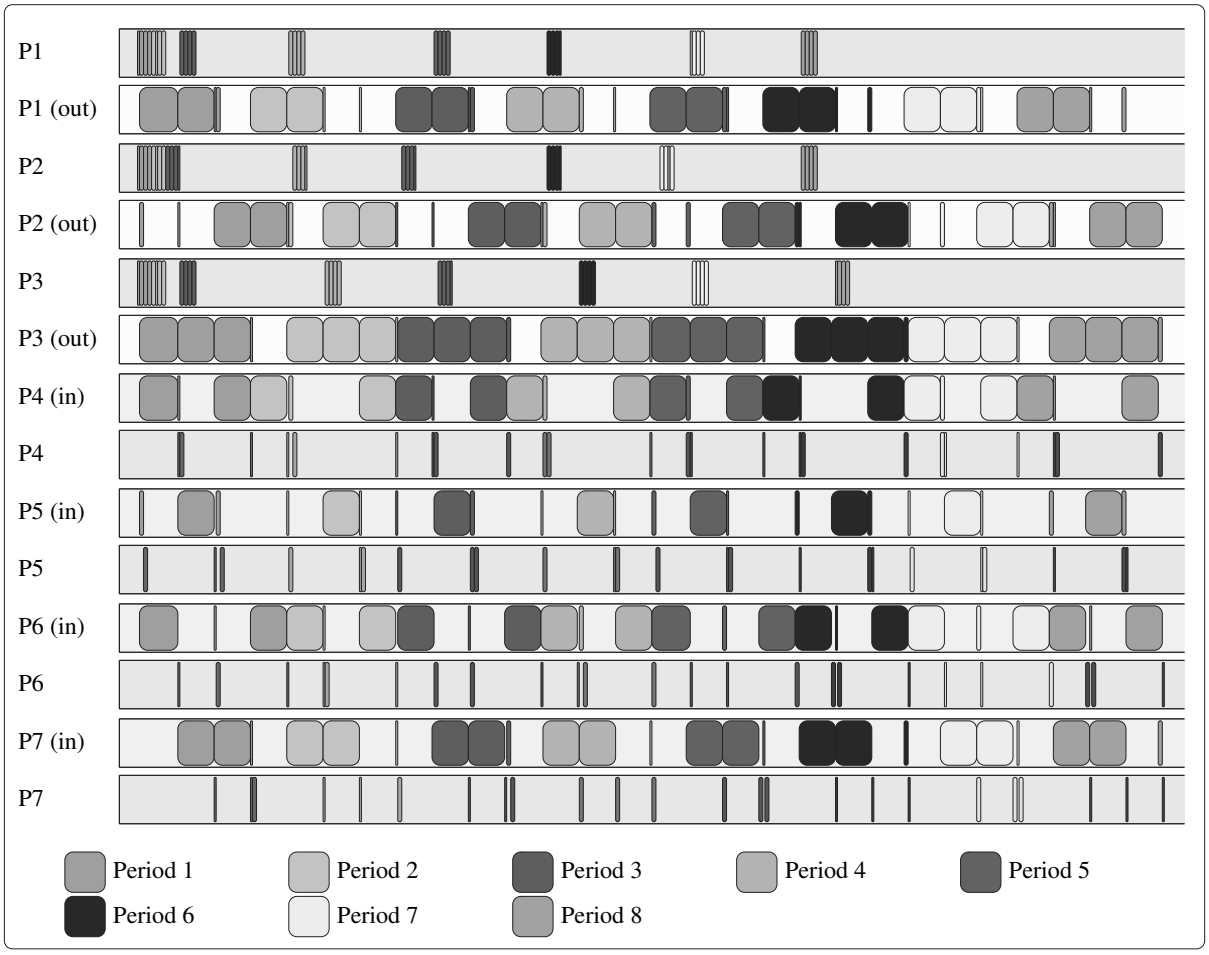


Figure 6.10: Gantt diagram of the first periods of Example B.

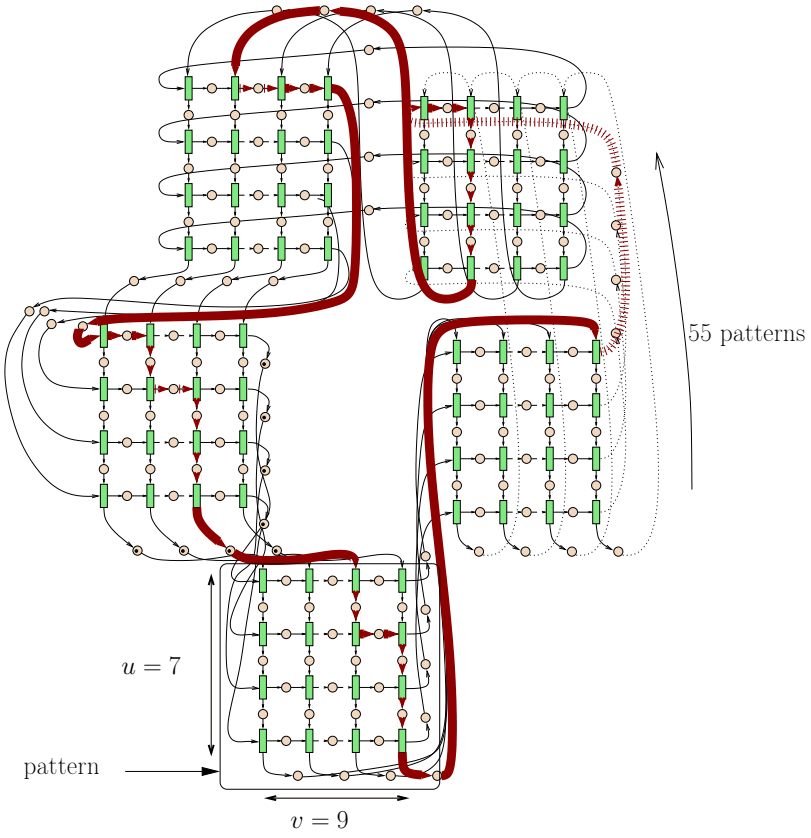


Figure 6.11: A complete connected component $\mathcal{G}$ (corresponding to Example C).

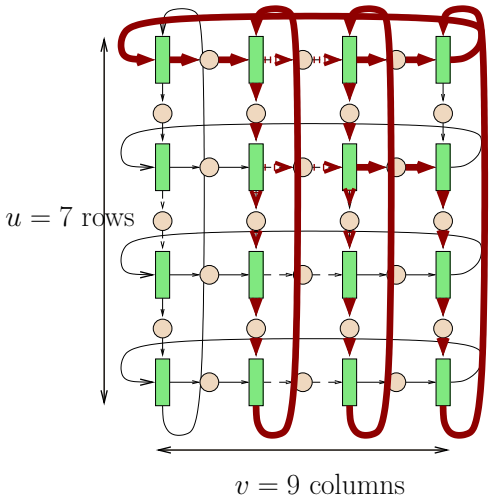


Figure 6.12: A single pattern $\mathcal{G}'$.

- if $0 \leq j < v$, then there is a place from x_{ij} to $x_{i(j+1)}$,
- there is a place from $x_{(u-1)j}$ to x_{0j} and from $x_{i(v-1)}$ to x_{i0}.

In Figure 6.12, we can see this graph $\mathcal{G}'$, corresponding the full graph shown in Figure 6.11.
We need some other definitions:

- If we consider a cycle $\mathcal{C}^a$ in $\mathcal{G}$, then by construction of P, the only way to pass through P is to enter by either the first column or the first line. Let k^a be the number of such entrances. Similarly, if we consider a cycle $\mathcal{C}^a$ in $\mathcal{G}'$, let k^a be the number of places $x_{(u-1)j} \rightarrow x_{0j}$ and $x_{i(v-1)} \rightarrow x_{i0}$.
- Let $\mathcal{L}^a$ be the sum of all transitions of a cycle $\mathcal{C}^a$.
- If $\mathcal{C}^a = \left(x_{i_0,j_0}^{k_0}, x_{i_1,j_1}^{k_1}, \ldots, x_{i_a,j_a}^{k_a}\right)$ is a cycle in $\mathcal{G}$, let $\mathcal{C}^b = (x_{i_0,j_0}, x_{i_1,j_1}, \ldots, x_{i_a,j_a})$ be its *projection* in $\mathcal{G}'$; by construction, the same place can appears many times in $\mathcal{C}^b$.
- A cycle $\mathcal{C}^a$ in $\mathcal{G}'$ can be dived into $\mathcal{G}$ to obtain a cycle $\mathcal{C}^b$ in $\mathcal{G}$. This transformation is shown in Figure 6.13.
- On the contrary, a cycle $\mathcal{C}^a$ in $\mathcal{G}$ can be projected on $\mathcal{G}'$ to obtain a cycle $\mathcal{C}^b$ in $\mathcal{G}'$. This transformation is shown in Figure 6.14.

Obviously, if $\mathcal{C}^a$ is a cycle in $\mathcal{G}$, then k^a is a multiple of p, the total number of patterns in $\mathcal{G}$. Now, by construction of the sub-TPN, there is a single token in each place between the last and the first pattern. Thus, the number of tokens in $\mathcal{C}^a$ is equal to k^a/p.

1. Let $\mathcal{C}^1$ be any critical cycle of $\mathcal{G}$. Its weight (or length) is $\mathcal{L}^1$, and the number of tokens is equal to k^1/p. Since $\mathcal{C}^1$ is critical, $\mathcal{L}^1 \times \frac{p}{k^1}$ is maximal.
2. Let $\mathcal{C}^2$ be the projection of $\mathcal{C}^1$ in $\mathcal{G}'$. By construction of $\mathcal{C}^2$, $k^2 = k^1$ and $\mathcal{L}^2 = \mathcal{L}^1$. However, there is no reason for $\mathcal{C}^2$ to be elemental. We split $\mathcal{C}^2$ into $\left(\mathcal{C}_1^2, \ldots, \mathcal{C}_{r_2}^2\right)$, where $\mathcal{C}_i^2$ is elemental. Moreover, we have $\sum_{i=1}^{r_2} \mathcal{L}_i^2 = \mathcal{L}^2$ and $\sum_{i=1}^{r_2} k_i^2 = k^2$.

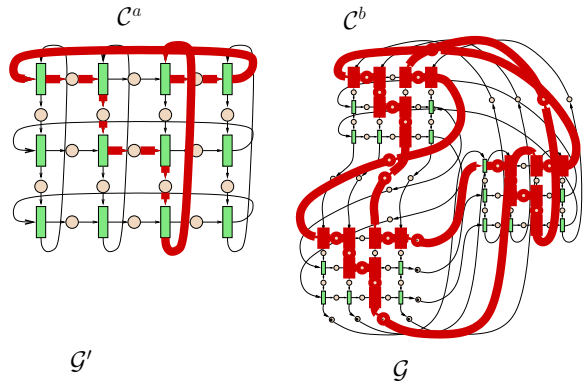


Figure 6.13: Diving $\mathcal{C}^a$ from $\mathcal{G}'$ to $\mathcal{G}$ to obtain $\mathcal{C}^b$.

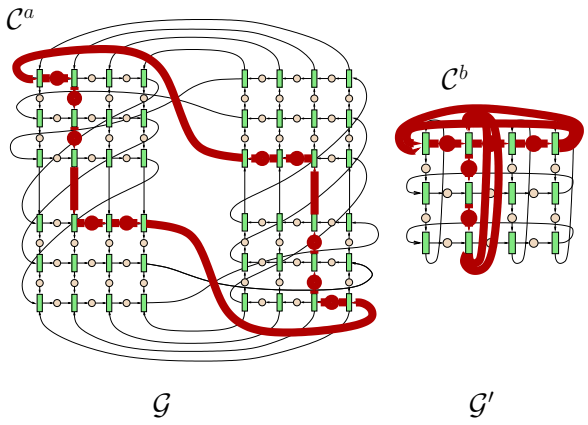


Figure 6.14: Projection of $\mathcal{C}^a$ from $\mathcal{G}$ to $\mathcal{G}'$ to obtain $\mathcal{C}^b$.

3. Let $\mathcal{C}^3$ be one of the $\mathcal{C}_i^2$ such that $\mathcal{L}^3/k^3 \geq \mathcal{L}^2/k^2$. Such an $\mathcal{C}_i^2$ exists, otherwise we have a contradiction: assume that we have

$$\forall i, \mathcal{L}_i^2/k_i^2 < \mathcal{L}^2/k^2$$

$$\Leftrightarrow \forall i, \mathcal{L}_i^2/k_i^2 < \frac{\sum_{j=1}^{r_2} \mathcal{L}_i^2}{\sum_{j=1}^{r_2} k_i^2}$$

$$\Leftrightarrow \forall i, \mathcal{L}_i^2 \sum_{j=1}^{r_2} k_j^2 < k_i^2 \sum_{j=1}^{r_2} \mathcal{L}_j^2.$$

We can sum these inequalities:

$$\Rightarrow \sum_{i=1}^{r_2} \left(\mathcal{L}_i^2 \sum_{j=1}^{r_2} k_j^2 \right) < \sum_{i=1}^{r_2} \left(\mathcal{L}_i^2 \sum_{j=1}^{r_2} k_j^2 \right)$$

$$\Leftrightarrow \left(\sum_{i=1}^{r_2} \mathcal{L}_i^2 \right) \left(\sum_{j=1}^{r_2} k_j^2 \right) < \left(\sum_{i=1}^{r_2} \mathcal{L}_i^2 \right) \left(\sum_{j=1}^{r_2} k_j^2 \right).$$

This last inequality is obviously wrong, showing that our $\mathcal{C}^3$ exists.

4. Let $\mathcal{C}^4$ be any elemental cycle of $\mathcal{G}'$, such that $\mathcal{L}^4/k^4$ is maximal. Since $\mathcal{C}^3$ is elemental, we have $\mathcal{L}^4/k^4 \geq \mathcal{L}^3/k^3$. Such a critical cycle can be found in time $O\left((uv)^3\right)$[11].

5. Let $\mathcal{C}^5$ the diving of $\mathcal{C}^4$ in $\mathcal{G}$. $\mathcal{C}^5$ is made of $c = \text{lcm}(p, k^4)/k^4$ copies of $\mathcal{C}^4$. Thus, we have $\mathcal{L}^5 = m\mathcal{L}^4$ and $k^5 = mk^4$. Finally, $\mathcal{L}^5/k^5 = \mathcal{L}^4/k^4$. Again, there is no reason for $\mathcal{C}^5$ to be an elemental cycle. We split $\mathcal{C}^5$ into $\left(\mathcal{C}_1^5, \ldots, \mathcal{C}_{r_5}^5\right)$, where $\mathcal{C}_i^5$ is elemental.

6. Let $\mathcal{C}^6$ be one of the $\mathcal{C}_i^5$ such that $\mathcal{L}^6/k^6 \geq \mathcal{L}^5/k^5$. As before, we can ensure that $\mathcal{C}^6$ exists, and $\mathcal{C}^6$ is elemental. Moreover, the number of tokens in $\mathcal{C}^6$ is equal to k^6/p.

7. Finally, we have:

$$\mathcal{L}^6/k^6 \geq \mathcal{L}^5/k^5 = \mathcal{L}^4/k^4 \geq \mathcal{L}^3/k^3 \geq \mathcal{L}^2/k^2 = \mathcal{L}^1/k^1.$$

Since p is positive, we have:

$$\mathcal{L}^6 p/k^6 \geq \mathcal{L}^5 p/k^5 = \mathcal{L}^4 p/k^4 \geq \mathcal{L}^3 p/k^3 \geq \mathcal{L}^2 p/k^2 = \mathcal{L}^1 p/k^1.$$

We know that $\mathcal{L}^1 p/k^1$ is maximal; since $\mathcal{C}^6$ is an elemental cycle, we have: $\mathcal{L}^6/k^6 = \mathcal{L}^1/k^1$ and thus,

$$\mathcal{L}^4/k^4 = \mathcal{L}^1/k^1.$$

We have shown that:

- $\mathcal{C}^4$ has the same critical weight as $\mathcal{C}^1$,
- $\mathcal{C}^4$ can be found without any knowledge on $\mathcal{G}$ nor $\mathcal{C}^1$,
- $\mathcal{C}^4$ is computed over $\mathcal{G}'$, which has a polynomial size.

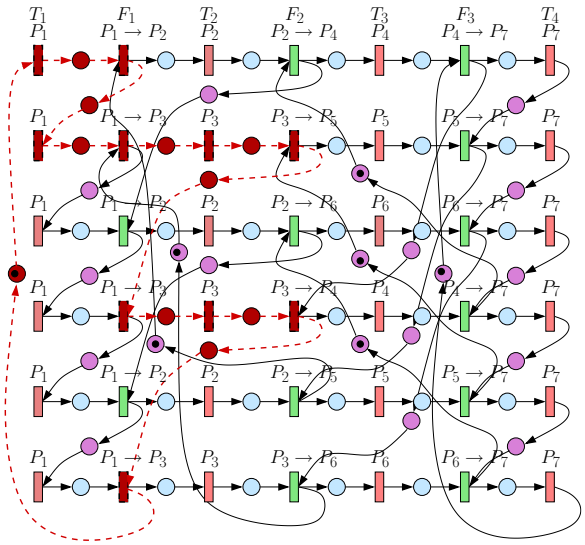


Figure 6.15: Complex critical cycles on Example A.

Hence, even if the sub-TPN has an exponential size, the length of its critical cycles can be found in polynomial time for each of its connected components.

Since we compute the critical cycles for all the p connected components, the total running time for the i-th communication is $\mathcal{O}\left((uv)^2 p\right) = \mathcal{O}\left((R_i R_{i+1})^3\right)$.

Thus, the total running time for all communications is $\mathcal{O}\left(\sum_{i=1}^{m-1}\left((R_i R_{i+1})^3\right)\right)$.

■

In Example A, a critical resource is the output port of P_1, whose cycle-time is equal to the period, 189. However it is possible to exhibit cases without critical resource: see for instance Example B presented in Figure 6.6. Its critical resource cycle-time is $\mathcal{M}_{\text{ct}} = 258.3$ and corresponds to the outgoing communications of P_3. It is strictly smaller than the actual period of the complete system, $\mathcal{T} = 291.7$. While the resource cycle-time can be computed by hand, the period of the system is given by ERS [57].

6.4.2 Strict One-Port model

Cycles in the TPN associated to the Strict One-Port model are more complex and less regular, since corresponding TPNs have backward edges. An example of such a cycle is shown in Figure 6.15. The intuition behind these backward edges is that a processor P_u cannot compute an instance of T_i before having completely sent the result F_i of the previous instance of T_i to the next processor P_v. Thus, P_u can be slowed by P_v. As for the Overlap One-Port model, there exist mappings for which all resources have idle times during a complete period. With the Strict One-Port model, this is the case for Example A, whose Gantt chart is presented in Figure 6.16: the critical resource is P_2, which has a cycle-time $\mathcal{M}_{\text{ct}} = 215.8$, strictly smaller than the period $\mathcal{T} = 230.7$.

Size (stages, processors)	Computation times	Communication times	#exp without critical resource / total
With overlap:			
(10, 20) and (10, 30)	between 5 and 15	between 5 and 15	0 / 220
(10, 20) and (10, 30)	between 10 and 1000	between 10 and 1000	0 / 220
(20, 30)	between 5 and 15	between 5 and 15	0 / 68
(20, 30)	between 10 and 1000	between 10 and 1000	0 / 68
(2, 7) and (3, 7)	1	between 5 and 10	0 / 1000
(2, 7) and (3, 7)	1	between 10 and 50	0 / 1000
Without overlap:			
(10, 20) and (10, 30)	between 5 and 15	between 5 and 15	14 / 220 (diff less than 9%)
(10, 20) and (10, 30)	between 10 and 1000	between 10 and 1000	0 / 220
(20, 30)	between 5 and 15	between 5 and 15	5 / 68 (diff less than 7%)
(20, 30)	between 10 and 1000	between 10 and 1000	0 / 68
(2, 7) and (3, 7)	1	between 5 and 10	10 / 1000 (diff less than 3%)
(2, 7) and (3, 7)	1	between 10 and 50	0 / 1000

Table 6.2: Numbers of experiments without critical resource.

6.5 Experiments

In Section 6.4, we have shown examples of mappings without any critical resource, i.e., whose period is larger than any resource cycle-time, for both communication models. We have conducted extended experiments to assess whether such situations are very common or not. Several sets of applications and platforms were considered, with between 2 and 20 stages and between 7 and 30 processors. All relevant parameters (processor speeds, link bandwidths, number of processors computing the same stage) were randomly chosen uniformly within the ranges indicated in Table 6.2. Finally, each experiment was run for both models. We compared the inverse of the critical resource cycle-time and the actual throughput of the whole platform. A grand total of $5,152$ different experiments were run. Table 6.2 shows that the cases without critical resources are very rare. In fact no such case was actually found with the Overlap One-Port model!

The computation times closely depends on the duplication factor of each stage: the computation of an example with 10 stages and 20 processors ranges from 2 to $150,000$ seconds on powerful machines such as a quadri-core server.

6.6 Conclusion

In this chapter, we have studied the throughput of streaming applications mapped on heterogeneous platforms. The major originality of this work, and also its major difficulty, is that we consider stage replication. Although this technique is classical in the literature, the computation of the throughput of such complex mappings had not been addressed yet (at the best of our knowledge). We have introduced TPNs (timed Petri nets) to determine the critical cycles of the mapping. 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 for both models. However, experiments show that such cases are very rare under the Overlap One-Port model. In addition, we have established the

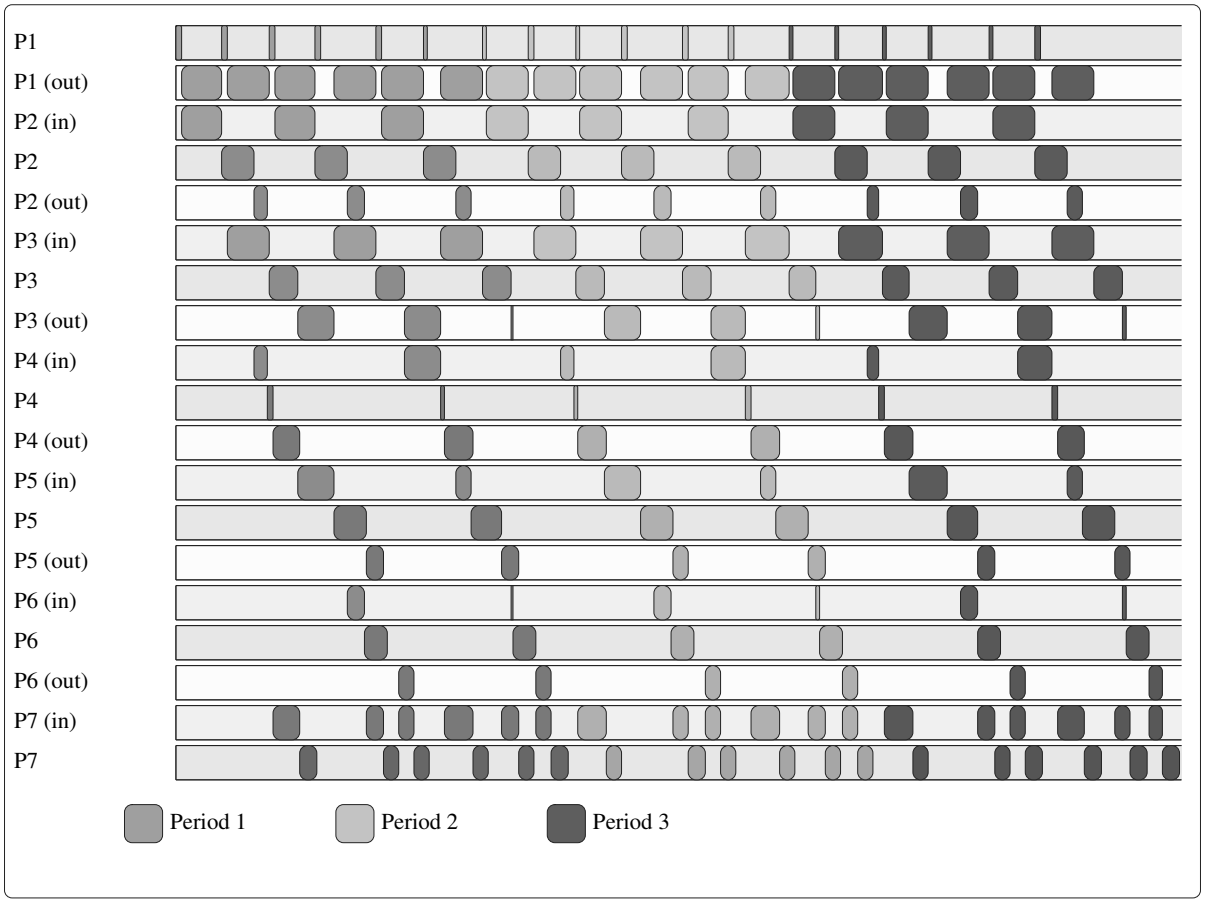


Figure 6.16: Gantt diagram of a schedule without critical resource.

polynomial complexity of the problem for this Overlap One-Port model, while it remains open for the Strict One-Port model.

This work was focused on static platforms, opening the way to future work on finding good schedules on dynamic platforms, whose speeds and bandwidths are modeled by random variables.

Chapter 7

Task graph scheduling on the Cell processor

7.1 Introduction

Contrarily to the previous chapters, in which we studied the deployment of applications on very large platforms such as computing grids, we now target a smaller platform: the Cell processor. This processor is an example of the emerging heterogeneous multi-core architectures, which can be made of homogeneous multi-core processors assisted by graphics processing units (GPUs), or by heterogeneous multi-core processors.

If homogeneous multi-core processors are now widespread, heterogeneous ones are quite recent, and softwares squeezing the most out of them are still rare. In this chapter, we study the Cell, which is made of a single, classical, PowerPC processing unit, and of up to eight smaller cores, dedicated to vectorial computing. As usual, we study the deployment of applications modeled by DAGs: we have many instances of the same DAG to schedule, and we aim at maximizing the throughput of the platform.

The Cell being the processor of game consoles like the Sony PlayStation 3, multimedia softwares are an obvious target of our work, but many scientific applications can also benefit from it, as soon as they are fine-grain parallel, streaming applications. Indeed, the usefulness of the Cell for scientific computing has lead IBM to sell server blades [56] based on two Cell processors. Due to its singularity, several solutions using a stream approach have been specifically developed for the Cell, like the StreamIt framework [51] or the DataCutter-Lite implementation [52]. Some other frameworks allow to handle communications and are well suited to matrix operations, like ALF [55], Sequoia [43], CellSs [21] or BlockLib [3].

We enforce the rule that all instances of the same task are mapped on the same resource: we use a single allocation, as explained in Section 4.1. This rule allows simpler flow control, and the strong memory constraints we have also incite us to enforce this rule: running the same task on different cores would lead to a useless duplication of variables. Splitting an application into fine-grain tasks is naturally done by associating a task to any function; however, dispatching these tasks to the available cores is a difficult problem, due to all the constraints.

We begin by describing in Section 7.2 the model of the Cell that we use to design our algorithms. Then we explain our optimal solution, obtained through the resolution of a mixed linear program in Section 7.3. In Section 7.4, we present the results of our preliminary experiments, before concluding this chapter in Section 7.5.

7.2 Modeling the Cell

In this section, we present our theoretical view of the Cell processor, as well as some useful notations, which were not presented in the general introduction in Section 3.2.

7.2.1 Processor model

As said in the introduction, the Cell is a heterogeneous multi-core processor jointly developed by Sony Computer Entertainment, Toshiba, and IBM, in which we can find the following components:

One Power core: also known as the PPE (standing for Power Processing Element) and respecting the Power ISA 2.03 standard, this core is two-way multithreaded. Its main role is to control the other cores, and to be used by the OS due to its similarity with existing processors. It has a 512-kB Level 2 cache and a 64-kB Level 1 cache, split into a 32-kB cache for the data and a 32-kB cache for the instructions. A SIMD instruction set, known as AltiVec or VMX, is also included in the PPE. We model multi-processor platforms by a single Cell processor with more Power cores. These Power cores are hereafter denoted by PPE_i.

Several Synergistic Processing Elements (SPE) cores: these cores constitute the main innovation of the Cell processor and are small 128-bit RISC processors specialized in simple precision, SIMD operations. These differences induce that some tasks are by far faster when processed on a SPE, while some other tasks can be slower, as stated by the unrelated machine. Each SPE has its own local memory (called *local store*) of size $mem = 256$ kB, and can access to other local stores and to the main memory only through explicit asynchronous DMA calls. While the current Cell model has eight SPEs, only six of them are available in the PS3, and IBM is working on models with up to 32 SPEs. Therefore, we have to consider any number of SPEs in our model.

The main XDR memory: only PPEs have a transparent access to it. The dedicated memory controller is integrated in the Cell and allows a fast access to the requested data. Since this memory is by far larger than the SPE's local stores, we consider it as always large enough. Thus, we do not take its size into account in the following.

An Element Interconnect Bus (EIB): this ring bus links all parts of the Cell to each other. The EIB has a bandwidth $BW = 200$ GB/s, and each component is connected to the EIB through a bidirectional interface, with a bandwidth $bw = 25$ GB/s.

All these components are displayed in Figure 7.1, and a more schematic image is shown in Figure 7.2. To simplify further equations, $PPEs = \{PE_0, \ldots, PE_{n_p-1}\}$ denotes the set of PPEs, while $SPEs = \{PE_{n_p}, \ldots, PE_{n_p+n_s-1}\}$ is the set of SPEs. Let n be the total number of cores, i.e., $n = n_p + n_s$.

Communication model. The bus is able to route all concurrent communications, and all communication elements are fully bidirectional. We use a bounded-multiport, linear communication cost model: a data of size S is sent or received in time S/b, where b is the bandwidth used for this communication, and the sum of incoming or outgoing communications of any element does not exceed its bandwidth. Memory accesses are counted as communications. Due to the limited

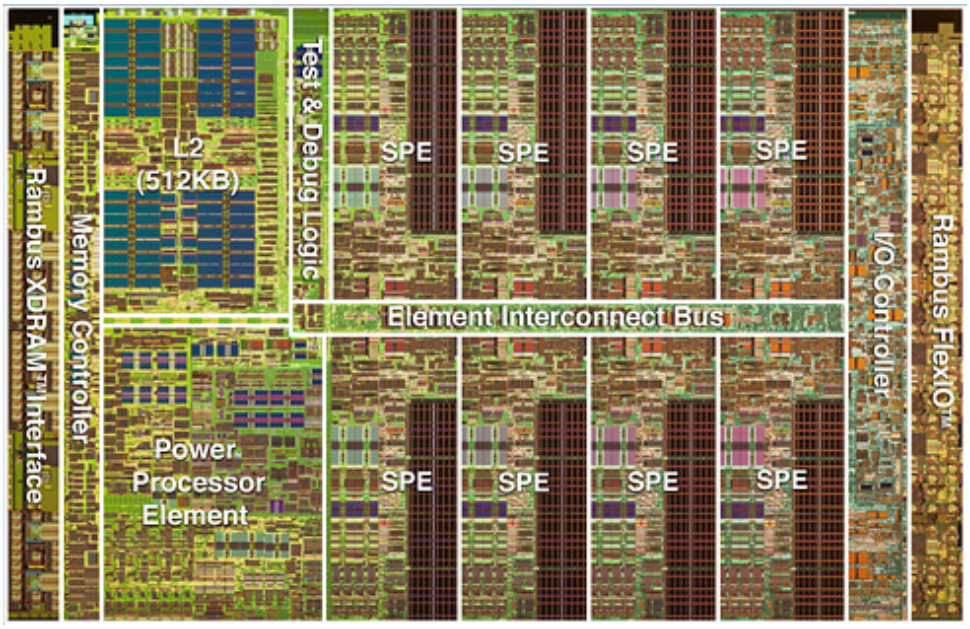


Figure 7.1: The complete Cell processor, and all its components.

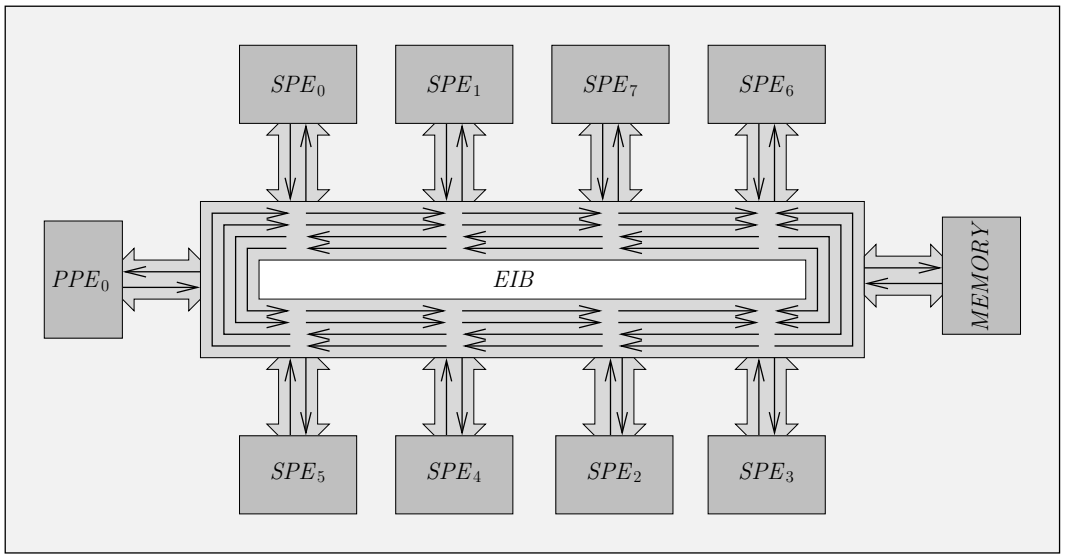


Figure 7.2: Theoretical view of the Cell.

size of the DMA stack on each SPE, each SPE can issue at most 16 simultaneous DMA calls, while the PPEs can issue at most eight simultaneous DMA calls to or from a given SPE. Since each core has a dedicated DMA engine, communications can be overlapped by computations.

7.2.2 Application model and schedule

The targeted applications are modeled by DAGs, as explained in Section 3.2.2. Thus, we consider a graph $G_A = (V_A, E_A)$. Due to the fine-grain model, we have to consider the difference between communications between two tasks T_k and T_l, which are denoted by $F_{k,l}$, and communications between a task T_k and the main memory. Indeed, we note $read_k$ the number of bytes read in memory by each instance of task T_k, and $write_k$ the number of bytes it writes to memory. Moreover, processing the t-th instance of task T_k requires the data corresponding to the t-th instance of a file $F_{l,k}$, but may also require the data corresponding to a given number of the next instances of the same file (let $peek_k$ be this number). In other words, T_k may need information on the near future (i.e., the next instances) before actually processing an instance. This requirement must be taken into account while designing algorithms, due to the strong memory constraints.

Since computing speeds of PPEs and SPEs are unrelated, $w_{PPE}(T_k)$ denotes the time required for a PPE to complete a single instance of T_k, while $w_{SPE}(T_k)$ is the time needed by a SPE to process a single instance of T_k. As all SPEs and all PPEs are identical, these two values suffice to fully describe the computation requirements.

Schedule reconstruction. Given a complete allocation $\sigma(G_A)$ of G_A on G_P, we use the same technique as in Section 3.3 to obtain the complete description of a valid periodic schedule π. Remind that a schedule is a function π associating:

- to each instance t of each task T_k, a starting computation time $\pi(T_k, t)$,
- to each instance t of each file $F_{k,l}$, a starting transfer time $\pi(F_{k,l}, t)$.

Since π is a periodic schedule of period $\mathcal{T}$, the starting time of the t-th instance of a given task T_k is given by a simple relation: $\pi(T_k, t) = \pi(T_k, 0) + t\mathcal{T}$; a similar relation holds true for the starting times of file communications. The whole execution is split into periods of length $\mathcal{T}$, and the u-th period of the schedule is the time interval between $(u-1)\mathcal{T}$ and $u\mathcal{T}$.

Due to dependency and peek constraints, we need to carefully select the first period, during which a task (or a communication) is executed.

Theorem 7.1. *Given a task graph G_A and a valid allocation scheme σ with a throughput $\rho = 1/\mathcal{T}$, we obtain a valid periodic schedule by applying the following rules:*

1. *If T_k has no incoming dependency, then its first instance is processed during the first period,*
2. *If the first instance of T_k is processed during the u-th period and if T_k produces a file $F_{k,l}$, then the first instance of $F_{k,l}$ is sent during the $(u+1)$-th period,*
3. *If T_l has a peek equal to $peek_l$ and requires files $F_{k_0,l}, \ldots, F_{k_j,l}, \ldots, F_{k_d,l}$, and if the first instance of $F_{k_j,l}$ is sent during the u_j-th period, then the first instance of T_l is processed during the $(\max_{0 \leq j \leq d}(u_j) + peek_l + 1)$-th period.*

Since there is no dependency between two communications or computations inside the same period, the order of the different tasks in a given period does not matter. By construction, this order is the same in every period.

Proof. During a time interval of length $\mathcal{T}$, any resource has to process at most a single instance of each task or file allocated to it. By definition of the period of the allocation, this duration is enough to process all these instances. Indeed, all the first instances of tasks without dependency can be executed during the first period, as required by rule (1).

If the t-th instance of task T_k is processed during the u-th period, then it is finished before the end of this period. Therefore, any file $F_{k,l}$ produced by T_k can be sent during the $(u+1)$-th period as expected by rule (2).

Let us consider a task T_l requiring files $F_{k_0,l}, \ldots, F_{k_j,l}, \ldots, F_{k_d,l}$ and with a peek $peek_l$. To be processed, the t-th instance of T_k requires all instances of these files between the t-th and the $(t + peek_k)$-th ones. Since the required instances of file $F_{k_j,l}$ are received from the u_j-th to the $(u_j + peek_l)$-th period, the t-th instance of task T_l can be executed during the $(\max_{0\leq j\leq d}(u_j) + peek_l + 1)$-th period as expected by rule (3).

Finally, the execution period of any instance of any task is well defined by these rules, and dependencies and peek constraints are respected. Once the order of execution within a given period is fixed (but it has no importance, since there is no dependency between tasks or communications within a period), we obtain a complete description of a valid periodic schedule. ■

$\mu(T_k)$ denotes the period during which the first instance of T_k is executed. Similarly, $\mu(F_{k,l})$ denotes the period of the communication of the first instance of file $F_{k,l}$.

Determining buffer sizes. Since SPEs have only 256 kB of local store, memory constraints on allocations are tight, and we need to precisely model them and to compute the size of the required buffers. Mainly for technical reasons, the code of the whole application is replicated to the local stores of all SPEs and to the memory shared across the PPEs. However, the buffers required for communications are allocated only on the nodes really processing the tasks. If T_l requires a file $F_{k,l}$ with a peek $peek_l$, then we allocate an incoming buffer of size $buff_{k,l} = (data_{k,l}.(\mu(T_l) - \mu(T_k)))$: the data of instance t sent by T_k are received during the $(\mu(T_k)+t)$-th period, but are used during the $(\mu(T_l) + t - 1)$-th period. Note that the peek number is taken into account during the determination of the $\mu(v)$ alues. To simplify the communication scheme, buffers have the same size on the sender and on the receiver.

7.2.3 NP-completeness of throughput optimization

We now formally define the decision problem associated to the problem of maximizing the throughput.

Definition 7.1 (Cell-Single-Alloc)**.** *Given a directed acyclic application graph G_A, a Cell processor model with at least two cores, and a bound B, is there an allocation with throughput $\rho \geq B$?*

Theorem 7.2. *Cell-Single-Alloc is NP-complete.*

Proof. We first have to prove that the problem belongs to NP, that is that we can check in polynomial time that the throughput of a given allocation is greater than or equal to B. Thanks to Section 3.3, we know that this check can be made through the evaluation of a simple formula; Cell-Single-Alloc is thus in NP.

To prove that Cell-Single-Alloc is NP-complete, we use a reduction from 2-Partition, known to be NP-complete [46]. Consider an instance $\mathcal{I}_1$ of 2-Partition, that is a set $\mathcal{I}$ of c positive integers $a_{i,1\leq i\leq c}$. The problem is to find a subset $\mathcal{J}$, such that:

$$\sum_{a_i\in\mathcal{J}} a_i = \sum_{a_i\in\mathcal{I}\setminus\mathcal{J}} a_i.$$

We construct an instance $\mathcal{I}_2$ of Cell-Single-Alloc:

The application DAG is a simple collection of independent tasks, made of $c+n-2$ tasks $T_1,\ldots,T_{c+n-2}$, with the same computation time when executed on a SPE or on a PPE. The computation time of task T_k is given by:

$$w_{PPE}(T_k) = w_{SPE}(T_k) = \begin{cases} a_k & \text{if } 1\leq k\leq c, \\ \frac{\sum_{a_i\in\mathcal{I}}}{2} & \text{if } c<k\leq c+n-2. \end{cases}$$

The throughput B to reach is set to $2/(\sum_{a_i\in\mathcal{I}} a_i)$. Obviously, this construction is polynomial in the size of the original instance $\mathcal{I}_1$.

In any solution of $\mathcal{I}_2$, the last $n-2$ tasks are mapped on $n-2$ cores, and no other task can be mapped on these cores without exceeding the bound $1/B$ on the computation time. Consequently, all the first c tasks are mapped on two cores. Without any loss of generality, we assume that these two cores are SPE_0 and SPE_1.

Consider any solution $\mathcal{J}$ of $\mathcal{I}_1$. If a_i belongs to $\mathcal{J}$, then we map T_i on SPE_0. Otherwise T_i is mapped on SPE_1. By construction, all tasks mapped on SPE_0 and SPE_1 can be executed within a period equal to $1/B$, and we have an allocation of the $n+c-2$ tasks providing a throughput B.

Reciprocally, consider any solution of $\mathcal{I}_2$. $\mathcal{J}$ is the set of all a_is, such that T_i is mapped on SPE_0. We have $\sum_{a_i\in\mathcal{J}} a_i \leq 1/B = (\sum_{a_i\in\mathcal{I}} a_i)/2$ and $\sum_{a_i\in\mathcal{I}\setminus\mathcal{J}} a_i \leq 1/B$. By definition of B, we have $\sum_{a_i\in\mathcal{J}} a_i = \sum_{a_i\in\mathcal{I}\setminus\mathcal{J}} a_i$.

Thus, finding an allocation with throughput greater than B is equivalent to finding a solution to the instance of 2-Partition, and the transformation from a solution to another one is polynomial.

Therefore, Cell-Single-Alloc is NP-complete.

■

7.3 A steady-state scheduling algorithm

We saw in the previous section that a valid allocation has many constraints to respect; if ignoring some limitations like the communication bandwidths only leads to poor performance, violating some memory constraints may lead to unfeasible solutions.

We aim at maximizing the throughput ρ of the processor using the solution of a linear program gathering all these constraints. Before writing the constraints to respect, we introduce two notations:

- $\alpha_i^k = \begin{cases} 1 & \text{if } T_k \text{ is mapped on core } PE_k, \\ 0 & \text{otherwise.} \end{cases}$
- $\beta_{i,j}^{k,l} = \begin{cases} 1 & \text{if, and only if, file } F_{k,l} \text{ is transfered from } PE_i \text{ to } PE_j, \\ 0 & \text{otherwise.} \end{cases}$

By definition, $\beta_{i,j}^{k,l}$ is equal to $\alpha_i^k \times \alpha_j^l$, but this redundancy between variables is required to write linear constraints. Constraints on these variables have different origins, as stated below:

The application structure:

- Each task is mapped on exactly one processor:

$$\forall T_k, \sum_{i=0}^{n-1} \alpha_i^k = 1.$$

- Given a dependency $F_{k,l}$, the core computing T_l must receive the corresponding data:

$$\forall F_{k,l}, \forall j, 0 \leq j < n, \sum_{i=0}^{n-1} \beta_{i,j}^{k,l} \geq \alpha_j^l.$$

- Given a dependency $F_{k,l}$, only the processor computing T_k is able to send the corresponding file to only one PE:

$$\forall F_{k,l}, \forall i, 0 \leq i < n, \sum_{j=0}^{n-1} \beta_{i,j}^{k,l} \leq \alpha_i^k.$$

The achievable throughput $\rho = 1/\mathcal{T}$:

- On a given PPE, all tasks must be completed within $\mathcal{T}$:

$$\forall i, 0 \leq i < n_p, \sum_{T_k} \alpha_i^k w_{PPE}(T_k) \leq \mathcal{T}.$$

- The same constraint holds true for the SPEs:

$$\forall i, n_p \leq i < n_s, \sum_{T_k} \alpha_i^k w_{SPE}(T_k) \leq \mathcal{T}.$$

- All incoming communications must be completed within $\mathcal{T}$:

$$\forall i, 0 \leq i < n, \sum_{T_k} \alpha_i^k read_k + \sum_{F_{k,l}} \sum_{0 \leq j < n, i \neq j} \beta_{j,i}^{k,l} \frac{data_{k,l}}{bw} \leq \mathcal{T}.$$

- All outgoing communications must be completed with $\mathcal{T}$:

$$\forall i, 0 \leq i < n, \sum_{T_k} \alpha_i^k write_k + \sum_{F_{k,l}} \sum_{0 \leq j < n, i \neq j} \beta_{i,j}^{k,l} \frac{data_{k,l}}{bw} \leq \mathcal{T}.$$

The hardware limitations of the Cell. The Cell has very specific constraints, especially on communications between cores. Even if SPEs are able to receive and send data while they are doing some computation, they are not multithreaded and the computation must be interrupted to initiate a communication (but the computation is resumed immediately after the initialization of the communication). There are two ways to transfer data from a core to another:

1. The sender writes data into the destination local store;

2. The receiver reads data from the source local store. This method is a bit faster, and we decide to use it.

Due to the absence of auto-interruption mechanism, the computation thread regularly checks the status of current communications. Moreover, any core has a limited number of available identifiers for DMA calls. Since it is hard to precisely control the communication sequence, we assume that all communications within a given period are simultaneous.

- All temporary buffers used by SPEs must fit into their local stores:

$$\forall i, n_p \leq i < n, \sum_{T_k} \left(\alpha_i^k \left(\sum_{F_{k,l}} buff_{k,l} + \sum_{F_{l,k}} buff_{l,k} \right) \right) \leq mem.$$

- Any SPE can perform at most 16 simultaneous incoming DMA calls

$$\forall i, n_p \leq i < n, \sum_{0 \leq j < n, i \neq j} \sum_{F_{k,l}} \beta_{j,i}^{k,l} \leq 16.$$

- Any PPE can perform at most eight simultaneous DMA calls from a given SPE:

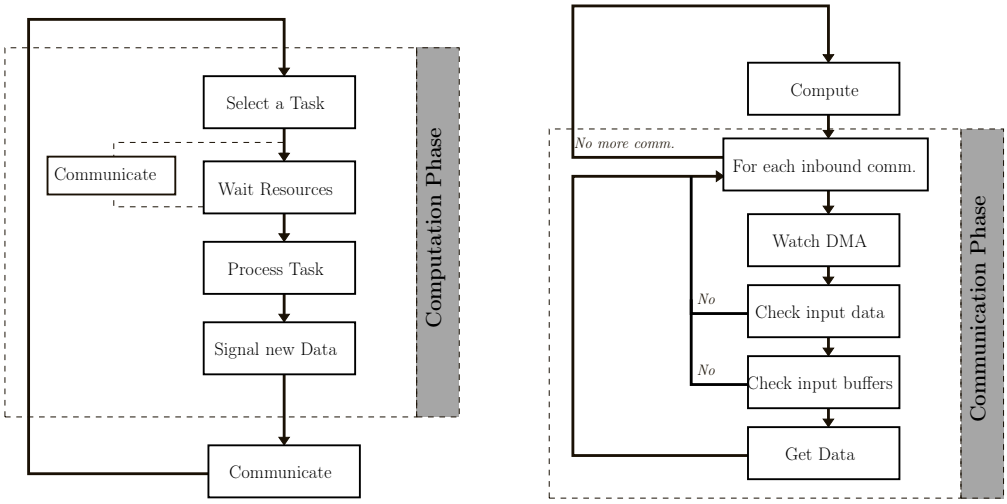
$$\forall i, 0 \leq i < n_p, \forall j, n_p \leq j < n, \sum_{F_{k,l}} \beta_{j,i}^{k,l} \leq 8.$$

These constraints form the following linear program:

$$\begin{cases} \text{MINIMIZE } \mathcal{T} \text{ UNDER THE CONSTRAINTS} \\ (7.1a) \quad \forall T_k, \quad \sum_{i=0}^{n-1} \alpha_i^k = 1 \\ (7.1b) \quad \forall F_{k,l}, \forall j, 0 \leq j \leq n-1, \quad \sum_{i=0}^{n-1} (\beta_{i,j}^{k,l}) \geq \alpha_j^l \\ (7.1c) \quad \forall F_{k,l}, \forall i, 0 \leq i \leq n-1, \quad \sum_{j=0}^{n-1} (\beta_{i,j}^{k,l}) \leq \alpha_i^k \\ (7.1d) \quad \forall i, 0 \leq i < n_p, \quad \sum_{T_k} (\alpha_i^k w_{PPE}(T_k)) \leq \mathcal{T} \\ (7.1e) \quad \forall i, n_p \leq i < n, \quad \sum_{T_k} (\alpha_i^k w_{SPE}(T_k)) \leq \mathcal{T} \\ (7.1f) \quad \forall i, 0 \leq i < n, \quad \alpha_i^k read_k + \sum_{F_{k,l}} \sum_{0 \leq j < n, j \neq i} (\beta_{j,i}^{k,l} \frac{data_{k,l}}{bw}) \leq \mathcal{T} \\ (7.1g) \quad \forall i, 0 \leq i < n, \quad \alpha_i^k write_k + \sum_{F_{k,l}} \sum_{0 \leq j < n, j \neq i} (\beta_{i,j}^{k,l} \frac{data_{k,l}}{bw}) \leq \mathcal{T} \\ (7.1h) \quad \forall i, n_p \leq i < n, \quad \sum_{T_k} \left(\alpha_i^k \left(\sum_{F_{k,l}} buff_{k,l} + \sum_{F_{l,k}} buff_{l,k} \right) \right) \leq mem \\ (7.1i) \quad \forall i, n_p \leq i < n, \quad \sum_{0 \leq j < n, i \neq j} \sum_{F_{k,l}} \beta_{j,i}^{k,l} \leq 16 \\ (7.1j) \quad \forall i, 0 \leq i < n_p, \forall j \neq i, 0 \leq j < n, \quad \sum_{F_{k,l}} \beta_{j,i}^{k,l} \leq 8 \end{cases} \tag{7.1}$$

7.4 Experiments

To assess the quality of both our model and our solution, we conduct several experiments. We used a real hardware platform: a Sony PlayStation 3. This game console is built around a single Cell processor, with 6 usable SPEs and a single Power core.



(a) Computation Phase. (b) Communication Phase.

Figure 7.3: Scheduler state machine.

Scheduling framework. Together with this hardware platform, we also needed a software framework to execute our schedules while handling communications. If there already exist some frameworks dedicated to stream applications [51, 52], none of them are able to deal with complex task graphs while allowing to statically select the mapping. Thus, we decided to develop one. Our scheduler only requires the description of an allocation as input. Even if it was designed to use the solution returned by the linear program (7.1), it can also use any mono-allocation schedule. We will now briefly describe our scheduler which is mainly divided into two main phases.

Computation phase: during which the scheduler selects a task and processes it.

Communication phase: during which the scheduler performs asynchronous communications.

These steps, depicted on Figure 7.3, are executed by every processing element. Moreover, since communications are supposed to be overlapped by computations, our scheduler cyclically alternates between these two phases.

The computation phase, which is shown on Figure 7.3(a), begins with the selection of a runnable task according to the provided schedule, then it waits for the required resources (input data and output buffers) to be available. If all required resources are available, the selected task is processed, otherwise, it skips to the communication phase. When new data is produced, the scheduler signals it to dependent processing elements.

The communication phase, depicted in Figure 7.3(b), aims at performing every incoming communication, most often by issuing DMA calls. Therefore, the scheduler begins by watching every previously issued DMA call in order to unlock the output buffer of the sender when data had been received. Then, the scheduler checks whether there is new incoming data. In that case, and if enough input buffers are available, it issues the proper Get command.

In our implementation, we had to deal with several issues:

- The Cell is, by nature, heterogeneous on several aspects:

  - The SPEs are 32 bits whereas the PPE is a 64-bit architecture.
  - Different communication mechanisms and constraints exist, depending on which processing elements are implied in the communication. For instance, we used the following intrinsics:
    * *mfc_get* for SPEs' inbound communications;
    * *spe_mfcio_put* for PPEs' inbound communications from SPEs;
    * *memcpy* for PPEs' inbound communications from main memory.

- Many variables need to be statically initialized in each local store before the execution of the first instances:
  - Information on tasks to execute on a given core,
  - Control variables, such as the addresses of all memory blocks or the identifiers of the currently processed instance,
  - Variables internally used by tasks,
  - Reception and transmission buffers.

  This initialization phase is mainly complicated by two trends:
  - Data structures, which are statically allocated, have varying sizes depending on the considered processing element (32/64 bits);
  - Runtime memory allocation.

The application. For our performance evaluation, we choose to fully implement a real-life application: the Vocoder. This application, whose task graph is presented in Figure 7.4, applies some modifications to any sound (often human voice) passed to its input. For instance, the Vocoder can be used to get a robot-like voice from a regular human voice, or to disguise a voice. To decrease the framework overhead, instances are aggregated into sets of 80 instances. Moreover, the original application was slightly modified to have balanced computations and communications, by artificially increasing the computation cost of some tasks.

This task graph is rather big (we have 141 tasks to schedule) and unbalanced since it has embarrassingly parallel parts as well as more sequential ones. It is therefore not straightforward to schedule this kind of graph on a heterogeneous parallel architecture like the Cell.

The mixed linear program is solved by Cplex [38] in less than two minutes, depending on the machine solving the linear program.

Reference heuristic. Since we have a large number of tasks, defining a schedule by hand is not feasible. Indeed, we use Algorithm 7 as a basis for comparisons. This algorithm tends to fill the local stores of the SPEs before allocating the remaining tasks to the first PPE. As experiments are run on a single-PPE platform, this limitation does not matter.

Experimental results. We compare the throughput obtained using our generated schedule to the one obtained using Algorithm 7, and the one placing all tasks on the PPE. We can see on Figure 7.5 that we outperform both other schedules: our throughput is more than 5.2 times higher with six SPEs than the one achieved by placing every task on the PPE, and 2.6 times higher than the one achieved by greedy.

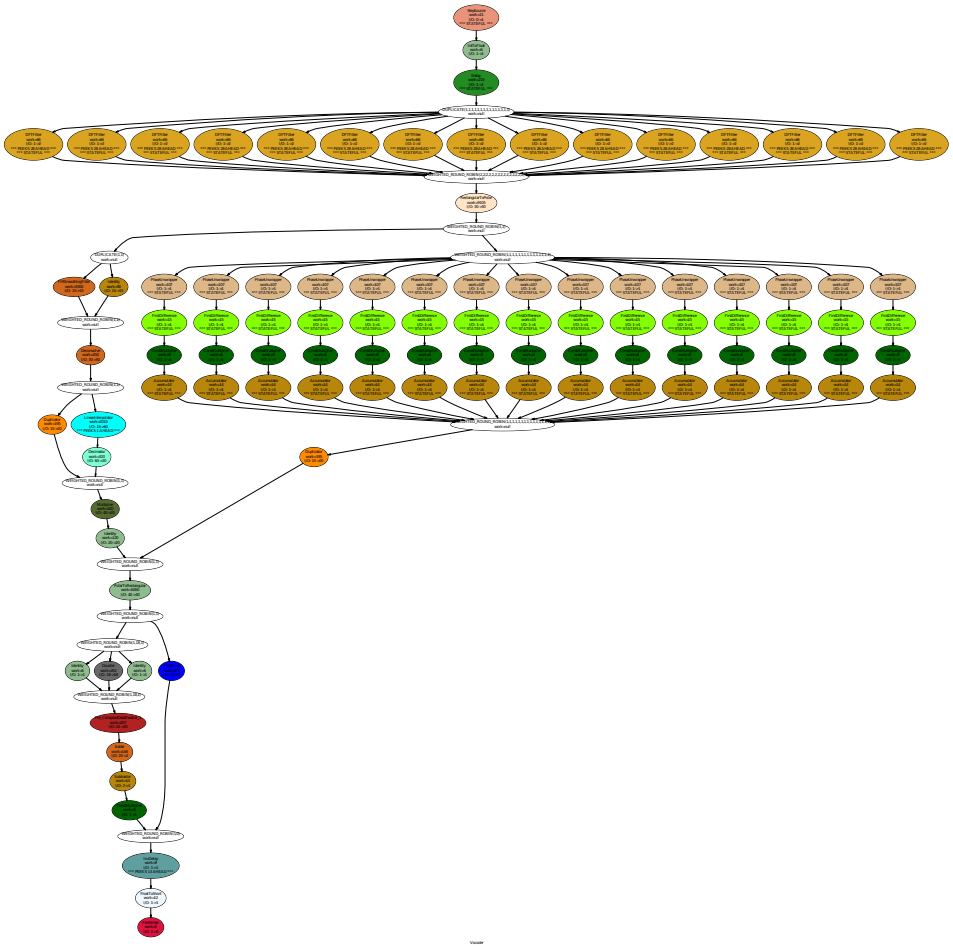


Figure 7.4: Task graph corresponding to the Vocoder application.

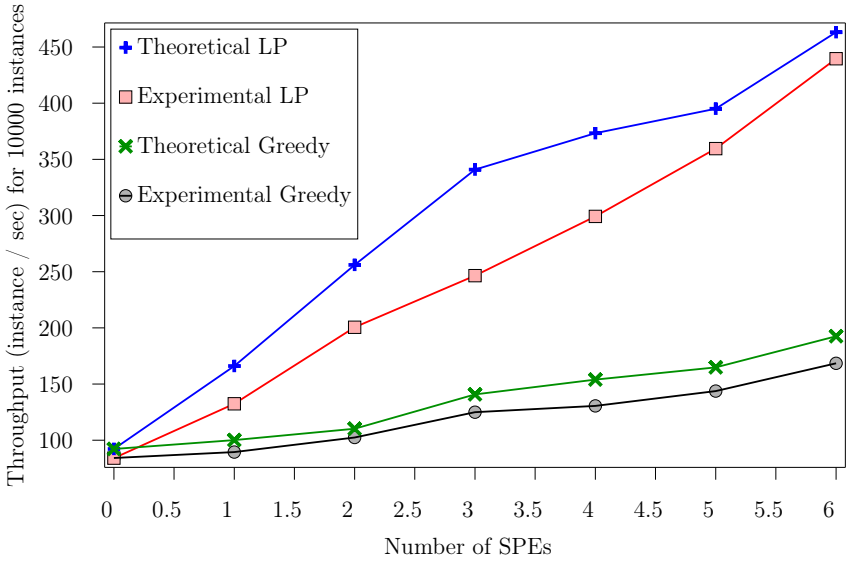


Figure 7.5: Throughput of Vocoder according to the number of SPEs used.

Algorithm 7: **Greedy**(G_P, G_A)

```
current_core ← SPE_0 ;
foreach T_k do
    if memory and DMA calls constraints are respected then
        mapping[T_k] ← current_core ;
    else
        if current_core ≠ SPE_{n_s−1} then
            current_core ← next SPE ;
        else
            current_core ← PPE_0 ;
        mapping[T_k] ← current_core ;
return mapping;
```

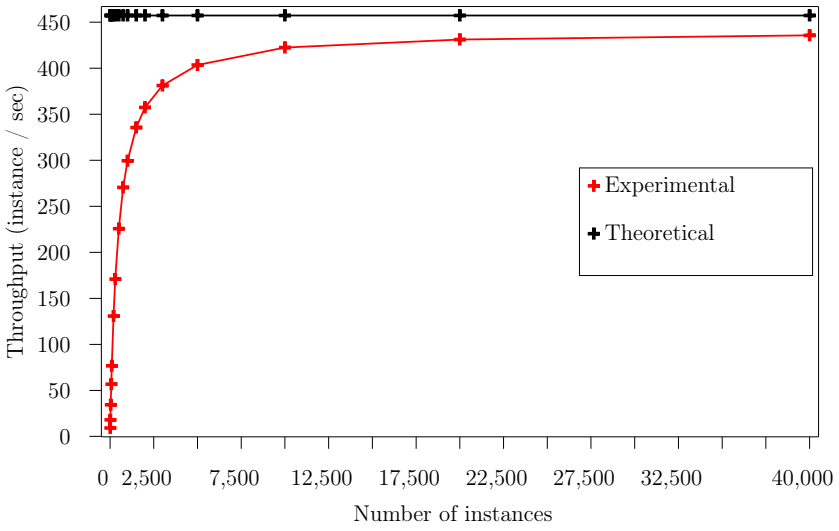


Figure 7.6: Throughput of Vocoder according to the number of instances used.

We now consider how our scheduling technique scales when the number of SPE is increased. We see that our approach scales better than the greedy algorithm. However, when comparing the theoretical throughput predicted by the linear program to the real throughput achieved by our application, there is a gap that could be explained mainly by the overhead introduced by our scheduling framework, and by our modeling of communications that might be too simple. These two trends still need to be precisely evaluated in future work.

Figure 7.6 shows that the steady state requires a quite large number of instances before being reached: the overall throughput is equal to 357 instances per second when using $2,000$ instances, while it becomes equal to 403 instances per second when using $5,000$ instances. Using more than $20,000$ instances does not significantly increase the overall throughput (431 instances per second, compared to the theoretical throughput equal to 457. This can be explained by the size of the task graph, and by the number of tasks peeking some data, increasing the finish time of the first instances.

7.5 Conclusion

Mainly due to the heterogeneity of the Cell and its strong hardware constraints, placing tasks on its different cores is a difficult problem, which has to be solved to efficiently use this processor. In this chapter, we first introduced a theoretical model of the Cell processor. We aim at scheduling a continuous flow of identical task graphs, and we use a steady-state approach using a single allocation of tasks on cores to maximize the throughput.

To assess the validity and usefulness of our approach, we implemented a complete framework, allowing to simply execute any steady-state schedule. Our method is compared to more classical scheduling techniques, and outperforms them in our experiments.

Chapter 8

Conclusion and Perspectives

8.1 Conclusion

In this thesis, we explored several problems, which are all focused on the scheduling of large numbers of instances of the same application. We aimed at optimizing the use of the computing platform by minimizing the computation time of these instances. Our main contributions are recalled in the following paragraphs.

Scheduling multiple divisible loads on a linear network

Our first contribution is a work on the well-known Divisible Load Theory. Concentrating on the scheduling of multiple divisible loads on a linear chain of processors, we intend to minimize their total completion time. Section 2.3 gives emphasis to the issues in the existing solution of this already studied problem, and this work gives in Section 2.4 a new and optimal solution. Moreover, we prove in Section 2.5 the intuitive assertion claiming that any optimal distribution of divisible loads on a linear chain of processors requires an infinite number of installments, as soon as a linear communication cost model is considered; this result remains valid if the linear chain of processors is replaced by a star-shaped platform.

Mono-allocation steady-state scheduling

Current computing grids are often as large as heterogeneous, and finding an efficient way to deploy many instances on them is a hard problem. Furthermore, even if a productive theoretical solution can be exhibited, this solution can be too complex to be actually implemented. The first issue was (partially) solved by changing the objective function: instead of minimizing the makespan which is the total completion time, we maximize the throughput, which is the average number of instances processed by the whole platform. However, an optimal solution maximizing the throughput is often made of a very large number of allocation schemes, and, thus, is not feasible.

This conclusion leads us to find in Chapter 4 clever solutions using a simpler allocation scheme, in order to allow a minimal program control while still reaching high throughputs. Although our solution presented in Section 4.3 is optimal, it is not feasible for large problems due to its high complexity in relation with the NP-completeness of our problem. Thus, we propose in Section 4.4 several heuristics, especially DELEGATE. Experiments presented in Section 4.5 show that this heuristic outperforms classical scheduling algorithms such as HEFT or Data-parallel and that it is close to the optimal mono-allocation solution.

Steady-state scheduling of dynamic bag-of-tasks applications

Bag-of-tasks applications, which are made of a set of independent tasks, are now frequently set up on dedicated computing grids, as well as on the so-called desktop grids. Simple star-shaped platforms, constituted of a single master processor owning all initial data and sending them to its worker, are commonly used to process these applications. In Chapter 5, we address the problem of finding schedules in this situation. The most employed schedulers are dynamic policies like on-demand or Round-Robin. However, while these strategies are versatile, they do not take into account the regularity of the application nor the peculiarities of the platform, downgrading the utilization of the physical resources.

This issue was already solved by adapted static schedulers when all instances of an application have identical characteristics, but we studied the problem when the instances of an application follow a given probability law in Section 5.3, and we proposed an ε-approximation in Subsection 5.3.2. Since this approximation requires knowledge about incoming instances (*off-line* model), we also describe an efficient heuristic in Subsection 5.3.3. Many experiments using the Simgrid framework prove the advantage of using such static schedules, even in the *online* case, when instances are only partially known.

Computing the throughput of replicated workflows

In Chapter 6, we consider a linear task graph, or *workflow*, of which we have a great number of copies to schedule on a heterogeneous platform. Unlike in the previous chapters, we assume that instances of a given task may be processed by distinct processors, and we apply a Round-Robin distribution of instances to processors. In other words, such a task is *replicated* on several processors. Two common communication models are analyzed. In the Overlap One-Port model, processors can process data while it is still both receiving and sending some other data in parallel, while a processor following the Strict One-Port model needs to do these operations sequentially.

Even if the mapping is given, computing the throughput is difficult, and we need to introduce complex models based on timed Petri nets in Section 6.3. These models enabled us to determine the throughput for both communication models, as established in Section 6.4. An amazing result is the absence of critical resource in some periodic schedules, any physical resource having idle times. Nonetheless, some simulations showed in Section 6.5 that these cases are infrequent.

Cell scheduling

While other chapters target large physical platforms, our last chapter, Chapter 7, is devoted to a single processor. This processor is the IBM Cell, a heterogeneous multi-core processor based on a single regular 32-bit PowerPC core and up to eight smaller 64-bit cores, called SPEs. All these cores are organized around a ring bus called EIB. These SPEs do not have cache memory, only a small local store, and the main memory is available through explicit DMA calls. Due to these features, optimizing a code for the Cell architecture is a hard work. In this chapter, we aim at scheduling a continuous flow of instances of a given task graph.

Our first contribution is a tractable theoretical model of this processor explained in Section 7.2. We also provide, in Section 7.3, an algorithm to schedule these instances while achieving a good throughput. To ease the use of periodic schedules and to assess the quality of our solution in Section 7.4, a new programming framework was created and a real-world application

was implemented. Preliminary results present an actual throughput close to that predicted by theory and better throughput than that returned by simpler greedy scheduling policies.

8.2 Perspectives

Few research topics can be considered as definitely closed, and those presented in this thesis make no exception. We expose hereinafter some guidelines for further research.

Mono-allocation schedules of task graphs on heterogeneous platforms

In Chapter 4, we look at periodic schedules made of a single allocation. Since all instances of a given task are processed on the same computing node, a schedule may require a lot of communications to use the largest possible number of processors. If these communications take a lot of time, some tasks may be duplicated on several processors to avoid several costly communications. Thus, even if this duplication increases the computation time, it may decrease the communication load of some critical links, especially in case of high communication-to-computation ratios. Moreover, another interest of this redundancy is fault tolerance. Tolerance to faults and maximization of throughput often are conflicting objectives. Thus, we may use bicriteria strategies to find a convenient trade-off.

Steady-state scheduling of bag-of-tasks applications

In Chapter 5, target platforms are constituted of related heterogeneous processors. However, due to the increasing heterogeneity of current computing grids, especially in the case of applications deployed through the BOINC framework, processors are more and more unrelated: a given processor which can quickly execute a given application may be quite slow to processor another one. Thus, we should extend our model to cope with this new source of heterogeneity. At a first glance, our solution should still be efficient in this case, contrarily to the On-Demand algorithm: even in the case of dominating computations, On-Demand is no more asymptotically optimal.

Replicated workflows on heterogeneous platforms

In Chapter 6, we expose how can be computed the throughput of replicated workflows on a heterogeneous platform. However, this computation requires the mapping to be known, and currently there is no clever approach to produce efficient mappings and schedules. Thus, since we can determine the effectiveness of a given schedule, we could search for efficient algorithms to take advantage of the platform. Due to the complexity of this problem, we should turn to polynomial heuristics rather than optimal schedules.

In this thesis, we deal with static applications, and we should move to dynamic ones, such that their instance data sizes and computation amounts conform to given probability laws. Another source of dynamicity comes from the platform itself. If processor speeds and link bandwidths are rather stable over time in the case of dedicated computing structures, many clusters or grids are shared by many users, implying considerable variations in resource loads resulting in variations in speeds and bandwidths. It would be interesting to investigate the computation of the throughput when tasks or resources are dynamic, and to see how to determine good schedules in this setting.

Task graph scheduling on the Cell processor

Although Chapter 7 is devoted to a single processor rather than a large computing platform, it is noteworthy to see how techniques originally designed to work on large-scale structures can also work at this scale. The solution we give in Section 7.3 is based on a mixed linear program and is not adapted to very large applications. Since our programming framework accepts any Steady-state schedule, we could elaborate some heuristics to decrease the exponential complexity of our heuristic while keeping a strong productivity. Moreover, our results are based on a single application, and we should apply our framework to several other applications to consolidate our results.

8.3 Final Remarks

During these last three years, the focus has been set on schedules specifically adapted to the execution on many identical, or at least similar, instances of the same job. Regularity played a key role in the solution design. Static schedules outperform dynamic ones because of this underlying regularity of the applications. We abandoned makespan minimization to deal with throughput optimization. However, even with this new "relaxed" objective, it turns out that computing the optimal solution is often too costly due to the inherent complexity of the problems, requiring the design of smart heuristics.

Appendix A

Bibliography

[1] M. Adler, Y. Gong, and A. Rosenberg. Asymptotically optimal worksharing in hnows: How long is "sufficiently long?". In *ANSS '03: Symposium on Simulation*, page 39, Washington, DC, USA, 2003. IEEE Computer Society Press.

[2] M. Adler, Y. Gong, and A. Rosenberg. On "exploiting" node-heterogeneous clusters optimally. *Theory of Computing Systems*, 42(4):465–487, 2008.

[3] M. Ålind, M. Eriksson, and C. Kessler. BlockLib: a skeleton library for Cell broadband engine. In *IWMSE '08: 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[4] D. Altilar and Y. Paker. An optimal scheduling algorithm for parallel video processing. In *ICMCS'98: International Conference on Multimedia Computing and Systems*, volume 0, page 245, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[5] D. Altilar and Y. Paker. Optimal scheduling algorithms for communication constrained parallel processing. In *Euro-Par 2002*, number 2400 in LNCS, pages 197–206, New York, NY, USA, 2002. Springer-Verlag.

[6] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS'67: American Federation of Information Processing Societies*, pages 483–485, New York, NY, USA, 1967. ACM Press.

[7] D. Anderson. BOINC: A system for public-resource computing and storage. In *GRID'04: 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society Press.

[8] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communication ACM*, 45(11):56–61, 2002.

[9] G. Ausiello, P. Crescenzi, V. Kann, G. Gambosi, and A. Marchetti-Spaccamela. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, January 2000.

[10] F. Baccelli, G. Cohen, and B. Gaujal. Recursive equations and basic properties of timed petri nets. *Journal of Discrete Event Dynamic Systems*, 1(4), 1992.

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

[12] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *IPDPS'02: International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[13] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: Results and open problems. Research report RR-4916, INRIA, 2003.

[14] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: results and open problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):207–218, 2005.

[15] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Scheduling strategies for mixed data and task parallelism on heterogeneous clusters. *Parallel processing letters*, 13(2), 2003.

[16] 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 *ISPDC'04: International Symposium on Parallel and Distributed Computing*, pages 296–302, July 2004.

[17] 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. Research report RR-2004-20, LIP, ENS Lyon, France, April 2004.

[18] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Pipelining broadcasts on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):300–313, 2005.

[19] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *International Journal of Foundations of Computer Science*, 16(2):163–194, 2005.

[20] O. Beaumont, L. Marchal, and Y. Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. In *HiPC'05: International Conference on High Performance Computing*, pages 123–132. Springer-Verlag, 2005.

[21] P. Bellens, J. Perez, R. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC'06: ACM/IEEE Super Computing Conference*, pages 5–5, Nov. 2006.

[22] A. Benoit, L. Marchal, and Y. Robert. Who needs a scheduler? Research report RR-2008-34, LIP, 2008.

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

[24] D. Bertsekas. *Constrained optimization and Lagrange Multiplier methods*. Athena Scientific, 1996.

[25] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithms for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[26] M. 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.

[27] V. Bharadwaj, D. Ghose, and V. Mani. Multi-installment load distribution in tree networks with delays. *IEEE Transactions on Aerospace and Electronic Systems*, 31(2):555–567, April 1995.

[28] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[29] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25:87–98, 1999.

[30] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, Secaucus, NJ, USA, 2001.

[31] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experimentations. In *UKSIM/EUROSIM'08: International Conference On Computer Modeling and Simulation*, volume 0, pages 126–131, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[32] S. Chan, V. Bharadwaj, and D. Ghose. Large matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: performance and simulation. *Mathematics and Computers in Simulation*, 58:71–92, 2001.

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

[34] 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.

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

[36] D. Coudert, H. Rivano, and X. Roche. A combinatorial approximation algorithm for the multicommodity flow problem. In *WAOA'03: Lecture Notes in Computer Science*, number 2909 in LNCS, pages 256–259. Springer-Verlag, 2003.

[37] J. Cowie, B. Dodson, E. Huizing, A. Lenstra, P. Montgomery, and J. Zayer. A world wide number field sieve factoring record: On to 512 bits. In *Advances in Cryptology — ASIACRYPT '96*, pages 382–394. Springer-Verlag, London, UK, 1996.

[38] ILOG CPLEX: High-performance software for mathematical programming and optimization. http://www.ilog.com/products/cplex/.

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

[40] R. Dick, D. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.

[41] M. Drozdowski. *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems*. PhD thesis, Instytut Informatyki Politechnika Poznanska, Poznan, 1997.

[42] C. Eck, J. Knobloch, L. Robertson, I. Bird, K. Bos, H. Brook, D. Düllmann, I. Fisk, D. Foster, B. Gibbard, C. Grandi, F. Grey, J. Harvey, A. Heiss, F. Hemmer, S. Jarp, R. Jones, D. Kelsey, M. Lamanna, H. Marten, P. Mato-Vila, F. Ould-Saada, B. Panzer-Steindel, L. Perini, Y. Schutz, U. Schwickerath, J. Shiers, and T. Wenaus. LHC computing Grid: Technical design report. version 1.06 (20 jun 2005). Technical Report CERN-LHCC-2005-024, CERN, June 2005.

[43] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. *SC'06: ACM/IEEE Super Computing Conference*, 0:4, 2006.

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

[45] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

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

[47] S. Genaud, A. Giersch, and F. Vivien. Load-balancing scatter operations for grid computing. *Parallel Computing*, 30(8):923–946, 2004.

[48] D. Ghose and T. Robertazzi, editors. *Special issue on* Divisible Load Scheduling. Cluster Computing, 6, 1, 2003.

[49] GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk/.

[50] R. Graham, D. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*. Wesley, 1994.

[51] X. Hang. A streaming computation framework for the Cell processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2007.

[52] T. Hartley and U. Catalyurek. A component-based framework for the Cell broadband engine. In *IPDPS'09: International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, june 2009. IEEE Computer Society Press.

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

[54] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *IPDPS'04: International Parallel and Distributed Processing Symposium*, Los Alamitos, CA, USA, 2004. IEEE Computer Society Press.

[55] IBM. Accelerated library framework. http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_alf_sdk30_1&S_TACT=105AGX16&S_CMP=EDU, 2007.

[56] IBM. IBM BladeCenter QS22. http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs21/index.html, 2009.

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

[58] L. Kachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[59] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551 – 563, 2003. Special Issue on Computational Grids.

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

[61] S. Larson, C. Snow, M. Shirts, and V. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[62] C. Lee and M. Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21:137–160, 1995.

[63] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *CCGrid'03: International Symposium on Cluster Computing and the Grid*, pages 138–145, May 2003.

[64] A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of biological requests. In *SPAA'06: Symposium on Parallelism in Algorithms and Architectures*, pages 103–112, New York, NY, USA, 2006. ACM Press.

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

[66] X. Li, B. Veeravalli, and C. Ko. Distributed image processing on a network of workstations. *International Journal of Computers and Applications*, 25(2):1–10, 2003.

[67] Pipealign. http://bips.u-strasbg.fr/PipeAlign/Documentation/.

[68] T. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, 2003.

[69] A. Rosenberg. Sharing partitionable workloads in heterogeneous NOWs: Greedier is not better. In *CLUSTER '01: 3rd IEEE International Conference on Cluster Computing*, page 124, Washington, DC, USA, 2001. IEEE Computer Society Press.

[70] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Euro-Par 2004*, number 3149 in LNCS, pages 173–182, New York, NY, USA, December 2004. Springer-Verlag.

[71] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

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

[73] B. Shirazi, A. Hurson, and K. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[74] 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.

[75] 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.

[76] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *SPAA'96: Symposium on Parallelism in Algorithms and Architectures*, pages 62–71, New York, NY, USA, 1996. ACM Press.

[77] F. Suter. Dag generation program. `http://www.loria.fr/~suter/dags.html`, 2009.

[78] K. Taura and A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW'00: Heterogeneous Computing Workshop*, pages 102–115, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[79] H. Topcuoglu, S. Hariri, and M. Wu. Task scheduling algorithms for heterogeneous processors. In *HCW'99: Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 1999. IEEE Computer Society Press.

[80] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, U. of California, Berkeley, May 2008.

[81] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Saddayappan, and J. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par 2007*, number 4641 in LNCS, pages 173–183, New York, NY, USA, 2007. Springer-Verlag.

[82] 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'08: International Conference on Parallel Processing*, pages 254–261, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[83] R. Wolski, N. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(10):757–768, 1999.

[84] H. Min Wong and B. Veeravalli. Scheduling divisible loads on heterogeneous linear daisy chain networks with arbitrary processor release times. *IEEE Transactions on Parallel and Distributed Systems*, 15(3):273–288, 2004.

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

[86] Q. Wu and Y. Gu. Supporting distributed application workflows in heterogeneous computing environments. In *ICPADS'08: 14th International Conference on Parallel and Distributed Systems*, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[87] Y. Yang and H. Casanova. Extensions to the multi-installment algorithm: Affine cost and output data transfers. Technical Report CS2003-0754, Dept. of Computer Science and Eng., Univ. of California, San Diego, July 2003.

[88] Y. Yang, H. Casanova, M. Drozdowski, M. Lawenda, and A. Legrand. On the complexity of multi-round divisible load scheduling. Research report RR-6096, INRIA, 2007. http://hal.inria.fr/inria-00123711.

[89] Y. Yang, K. van der Raadt, and H. Casanova. Multiround algorithms for scheduling divisible loads. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1092–1102, 2005.

Appendix B

Publications

The publications are listed in reverse chronological order.

Articles in international refereed journals and book chapter

[A1] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Divisible load scheduling. In *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2008. To appear.

[A2] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Comments on "design and performance evaluation of load distribution strategies for multiple loads on heterogeneous linear daisy chain networks". *Journal of Parallel and Distributed Computing*, 68(7):1021–1031, 2008.

Articles in international refereed conferences

[B1] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2009.

[B2] Anne Benoit, Matthieu Gallet, Bruno Gaujal, and Yves Robert. Computing the throughput of replicated workflows on heterogeneous platforms. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP'09)*, 2009.

[B3] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Allocating series of workflows on computing grids. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, 2008.

[B4] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling multiple divisible loads on a linear processor network. In *Proceedings of the 13rd IEEE International Conference on Parallel and Distributed Systems (ICPADS'07)*, 2007.

[B5] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling communication requests traversing a switch: complexity and algorithms. In *Proceedings of the 15th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'2007)*, pages 39–46. IEEE Computer Society Press, 2007.

Research reports

[C1] Matthieu Gallet, Anne Benoit, Yves Robert, and Bruno Gaujal. Computing the throughput of replicated workflows on heterogeneous platforms. Research report RR-2009-08, ENS Lyon, 2009.

[C2] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Allocating series of workflows on computing grids. Research report RR-6603, LIP, ENS Lyon, 2008.

[C3] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling multiple divisible loads on a linear processor network. Research report RR-6235, INRIA, 2007.

[C4] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Comments on "design and performance evaluation of load distribution strategies for multiple loads on heterogeneous linear daisy chain networks". Research report RR-6123, INRIA, 2007. Also available as LIP RR-2007-07.

[C5] Matthieu Gallet, Yves Robert, and Frédéric Vivien. Scheduling communication requests traversing a switch: complexity and algorithms. Research report RR-2006-25, LIP, ENS Lyon, 2006. Also available as LIP research report.

Appendix C

Notations

Chapter 1: Introduction to the Divisible Load Theory

α_i	Fraction of the total load allocated to processor P_i.
c	Time needed by any worker to receive a unit-size load (in case of homogeneous platform).
c_i	Time needed by worker P_i to receive a unit-size load.
C	Communication latency in case of homogeneous platform.
C_i	Communication latency of processor P_i.
M	The master processor, equal to P_0.
n	Number of processors in the system.
n_i	Number of tasks processed by processor P_i.
P_i	Processor i, where $i = 0, \ldots, n$.
R	Computation-communication ratio: $R = w/c$.
T	Total completion time of the system.
T_i	Completion time of processor P_i.
W_{total}	Number of tasks to process on the workers.
w	Time needed by any worker to process a unit-size load (in case of homogeneous platform).
W	Computation latency, in case of homogeneous platform.
w_i	Time required by processor P_i to process a single task.
W_i	Computation latency of processor P_i.

Chapter 2: Divisible Load Theory and Linear chains

bw_i	Bandwidth used by P_i to transmit a load to P_{i+1}.
β_l	Fraction $\gamma_2^l(2)$ corresponding to the l-th installment of the second load processed by P_2.
$Comm_{i,k,j}^{end}$	End time of communication from processor P_i to processor P_{i+1} for the j-th installment of the k-th load.
$Comm_{i,k,j}^{start}$	Start time of communication from processor P_i to processor P_{i+1} for the j-th installment of the k-th load.
$Comp_{i,k,j}^{end}$	End time of computation on processor P_i

	for the j-th installment of the k-th load.
$Comp_{i,k,j}^{start}$	Start time of computation on processor P_i for the j-th installment of the k-th load.
$\gamma_i^j(k)$	Fraction of the k-th load computed on processor P_i during the j-th installment.
λ	Time taken by P_1 and P_2 to process a unit-size load.
m	Total number of loads to process in the system.
n	Number of processors in the system.
Q_k	Total number of installments for k-th load.
P_i	Processor i, where $i = 1, \ldots, n$.
s_i	Speed of processor P_i.
τ_i	Availability date of P_i (time at which it first becomes available for processing the loads).
$V_{comm}(k)$	Volume of data for the k-th load.
$V_{comp}(k)$	Volume of computation for the k-th load.

Chapter 3: Introduction to steady-state scheduling

Most of these notations are also used in Chapters 4 – 7.

$bw_{i,j}$	Bandwidth of link $P_i \rightarrow P_j$.
E_A	Set of dependencies between tasks.
E_P	Set of communication links between processors.
$data_{k,l}$	Size of the file $F_{k,l}$.
$F_{k,l}$	Dependency between task T_k and task T_l, materialized by a file.
$F_{k,l}^u$	u-th instance of file $F_{k,l}$.
G_A	Directed Acyclic Graph model of the considered application, we have $G_A = (V_A, E_A)$.
G_A^u	u-th instance of the complete application.
G_P	Graph model of the computing platform, with $G_P = (V_P, E_P)$.
m	Number of tasks in the considered task graph.
n	Number of available processors.
P_i	Processor i, where $i = 1, \ldots, n$.
$P_i \rightarrow P_j$	Communication link between P_i and P_j.
$\sigma(G_A^u)$	Allocation of a single instance of the application graph G_A.
ρ	Average number of task graphs processed by the platform.
T_k	Task k, where $k = 1, \ldots, m$.
T_k^u	u-th instance of task T_k.
$\mathcal{T}$	Period of a periodic schedule.
V_A	Set of tasks, with $V_A = (T_1, \ldots, T_m)$.
V_P	Complete set of processors, with $V_P = (P_1, \ldots, P_n)$.

Chapter 4: Mono-allocation schedules of task graphs on heterogeneous platforms

bw_q^{in}	Maximum incoming bandwidth of processor P_q.
bw_q^{out}	Maximum outgoing bandwidth of processor P_q.

$f_{i,j}^{k,l}$	Average number of files $F_{k,l}$ traversing the link $P_i \to P_j$, where $P_i \to P_j$ belongs to any path $P_i \rightsquigarrow P_j$.
$P_q \rightsquigarrow P_r$	Any path from processor P_q to processor P_r, made of a set of adjacent links from P_q to P_r.
t_q^{comp}	Time passed by processor P_q to the execution of a single instance of G_A.
t_q^{in}	Time passed by processor P_q to the receptions during the execution of a single instance of G_A.
t_q^{out}	Time passed by processor P_q to the outgoing communications during the execution of a single instance of G_A.
$t_{q,r}$	Average occupation of $P_q \to P_r$ during the execution of a single instance of G_A.
$w_{i,k}$	Time needed by P_i to execute a single instance of task T_k.
$x_{q,r}^{k,l}$	Binary variable, equal to 1 if, and only if, file $F_{k,l}$ is sent from processor P_q to processor P_r.
y_q^k	Binary variable, equal to 1 if, and only if, task T_k is processed on processor P_q.

Chapter 5: Steady-state scheduling of dynamic bag-of-tasks applications

bw_0	Network card capacity of the master processor P_0, which initially owns all data.
bw_i	Bandwidth limit of worker P_i.
γ_q^k	Bounds of intervals of computation amount for application T_k $\left(\gamma_q^k = \min_{comp}^k (1+\varepsilon)^q\right)$.
δ_q^k	Bounds of data sizes intervals for application type T_k $\left(\delta_q^k = \min_{comm}^k (1+\varepsilon)^r\right)$.
ε	Desired precision of our approximation.
$I_{q,r}^k$	Cross product of a data size interval $\left(I_{q,r}^k = \left[\gamma_q^k; \gamma_{q+1}^k\right] \times \left[\delta_r^k; \delta_{r+1}^k\right]\right)$.
$\max_{comm}^k$	Upper bound on the data sizes of T_k instances.
$\max_{comp}^k$	Upper bound on the amount of computations of T_k instances.
$\min_{comm}^k$	Lower bound on the data sizes of T_k instances.
$\min_{comp}^k$	Lower bound on the amount of computations of instances of application T_k.
N^k	First instances of application T_k $\left(N^k = \left\lfloor \frac{\pi_k}{\pi_1} N^1 \right\rfloor\right)$.
$n_{q,r}^k$	Actual number of T_k instances in interval $I_{q,r}^k$ among N^k first ones.
$p_{q,r}^k$	Probability of an instance of application T_k to be in an interval $I_{q,r}^k$.
π_k	Priority of application T_k.
Q^k	Number of intervals of computation amount of application T_k $\left(Q^k = 1 + \left\lfloor \frac{\ln\left(\frac{\max_{comp}^k}{\min_{comp}^k}\right)}{\ln(1+\varepsilon)} \right\rfloor\right)$.

R^k	Number of data size intervals of application type T_k $\left(R^k = 1 + \left\lfloor \frac{\ln\left(\frac{\max^k_{comm}}{\min^k_{comm}}\right)}{\ln(1+\varepsilon)} \right\rfloor\right)$.
s_i	Speed of processor P_i.
$T_{k,q,r}$	Virtual application made of instances of T_k in the $I^k_{q,r}$ interval.
ρ^k	Average number of instances of application T_k processed by the whole platform.
ρ^k_i	Average number of instances of application T_k processed by processor P_i.
$\rho^k_{i,q,r}$	Contribution of processor P_i to the throughput of instances of application T_k in the interval $I^k_{q,r}$.
$V_{comm}(k)$	Volume of data for the k-th application type T_k.
$V_{comp}(k)$	Volume of computation for the k-th application T_k.
X^k_{comm}	Random variable describing the data size of instances of application T_k.
X^k_{comp}	Random variable describing the amount of computation of instances of application T_k.

Chapter 6: Computing the throughput of replicated workflows

$C_{\text{exec}}(i)$	Cycle-time of processor P_i.
$C_{\text{in}}(i)$	Reception time of processor P_i.
$C_{\text{out}}(i)$	Transmission time of processor P_i.
$C_{\text{comp}}(i)$	Computation time of processor P_i.
F_k	Dependency between T_k and T_{k+1}.
$\mathcal{L}(\mathcal{C})$	Length of cycle $\mathcal{C}$, i.e., the sum of the time of its transitions.
$\mathcal{M}_{\text{ct}}$	Maximum cycle-time, over all processors ($\mathcal{M}_{\text{ct}} = \max_{1\leq i\leq n} C_{\text{exec}}(i)$).
R	Number of different paths followed by the instances in the system.
R_k	Replication factor of stage T_k.
$t(\mathcal{C})$	Number of tokens present in cycle $\mathcal{C}$.
Tr^j_i	i-th transition of the j-th row. Tr^j_{2i} corresponds to the computation of T_i. Tr^j_{2i+1} corresponds to the transmission of a file F_i.
$V_{comm}(k)$	Size of the dependency F_k.
$V_{comp}(k)$	Computation volume of stage T_k.

Chapter 7: Task graph scheduling on the Cell processor

α^k_i	Binary variable, equal to 1 if, and only if, task T_k is processed on processing element PE_i.
$\beta^{k,l}_{i,j}$	Binary variable, equal to 1 if, and only if, file $F_{k,l}$ is sent from element P_i to element P_j.
$buff_{k,l}$	Size of the buffer used for transmission of $F_{k,l}$.
bw	Bandwidth of the link between the EIB and one of the Cell component.
BW	Bandwidth of the EIB ring bus.
mem	Size of the local store of each SPE (currently 256 KB).

PE_i	i-th processing element (including PPEs as well as SPEs).
$peek_k$	Number of further instances of T_k required for processing a single one.
PPE_i	i-th PPE core.
$read_k$	Number of bytes read in memory before processing an instance of T_k.
SPE_i	i-th SPE core.
$\mu(T_k)$	Period during which the first instance of task T_k is processed.
$write_k$	Number of bytes written in memory after the execution of an instance of T_k.
$w_{PPE}(T_k)$	Time required by a PPE to compute an instance of task T_k.
$w_{SPE}(T_k)$	Time required by a SPE to compute an instance of task T_k.

Résumé :
Les travaux présentés dans cette thèse portent sur l'ordonnancement d'applications sur des plate-formes hétérogènes à grande échelle. Dans la mesure où le problème général est trop complexe pour être résolu de façon exacte, nous considérons deux relaxations.

Tâches divisibles : La première partie est consacrée aux tâches divisibles, qui sont des applications parfaitement parallèles et pouvant être arbitrairement subdivisées pour être réparties sur de nombreux processeurs. Nous cherchons à minimiser le temps de travail total lors de l'exécution de plusieurs applications aux caractéristiques différentes sur un réseau linéaire de processeurs, sachant que les données peuvent être distribuées en plusieurs tournées. Le nombre de ces tournées étant fixé, nous décrivons un algorithme optimal pour déterminer précisément ces tournées, et nous montrons que toute solution optimale requiert un nombre infini de tournées, résultat restant vrai sur des plate-formes non plus linéaires mais en étoile. Nous comparons également notre méthode à des méthodes déjà existantes.

Ordonnancement en régime permanent : La seconde partie s'attache à l'ordonnancement de nombreuses copies du même graphe de tâches représentant une application donnée. Au lieu de chercher à minimiser le temps de travail total, nous optimisons uniquement le cœur de l'ordonnancement. Tout d'abord, nous étudions des ordonnancements cycliques de ces applications sur des plate-formes hétérogènes, basés sur une seule allocation pour faciliter leur utilisation. Ce problème étant NP-complet, nous donnons non seulement un algorithme optimal, mais également différentes heuristiques permettant d'obtenir rapidement des ordonnancements efficaces. Nous les comparons à ces méthodes classiques d'ordonnancement, telles que HEFT.
Dans un second temps, nous étudions des applications plus simples, faites de nombreuses tâches indépendantes, que l'on veut exécuter sur une plate-forme en étoile. Les caractéristiques de ces tâches variant, nous supposons qu'elles peuvent être modélisées par des variables aléatoires. Cela nous permet de proposer une ε-approximation dans un cadre clairvoyant, alors que l'ordonnanceur dispose de toutes les informations nécessaires. Nous exposons également des heuristiques dans un cadre non-clairvoyant. Ces différentes méthodes montrent que malgré la dynamicité des tâches, il reste intéressant d'utiliser un ordonnancement statique et non des stratégies plus dynamiques comme On-Demand.
Nous nous intéressons ensuite à des applications, dont plusieurs tâches sont répliquées sur plusieurs processeurs de la plate-forme de calcul afin d'améliorer le débit total. Dans ce cas, même si les différentes instances sont distribuées aux processeurs tour à tour, le calcul du débit est difficile. Modélisant le problème par des réseaux de Petri temporisés, nous montrons comment le calculer, prouvant également que ce calcul peut être fait en temps polynomial avec le modèle Strict One-Port.
Enfin, le dernier chapitre est consacré à l'application de ces techniques à un processeur multi-cœur hétérogène, le Cell d'IBM. Nous présentons donc un modèle théorique de ce processeur ainsi qu'un algorithme d'ordonnancement adapté. Une implémentation réelle de cet ordonnanceur a été effectuée, permettant d'obtenir des débits intéressants tout en simplifiant l'utilisation de ce processeur et validant notre modèle théorique.

Mots-clés :
Tâches divisibles, réseaux linéaires, multi-tournées, ordonnancement en régime permanent, plate-forme en étoile, maître-esclave, modèles stochastiques, réseaux de Petri temporisés, heuristiques, programmes linéaires, plate-formes hétérogènes, maximization du débit.

Abstract:
This thesis mainly deals with the mapping and the scheduling of applications on large heterogeneous platforms. As the general scheduling problem is untractable, we consider two relaxations which apply to specific problems.

Divisible load scheduling: Divisible loads are perfectly parallel applications, which can be split into chunks of arbitrary sizes to be distributed to many workers. We focus our attention on scheduling several divisible loads with different characteristics on linear networks of processors, in order to minimize the total processing time. This distribution may be done using several installments. Given a number of installments, we expose an algorithm giving an optimal distribution of loads on processors, and we compare it to a pre-existing solution. Moreover, we show that any optimal distribution uses an infinite number of installments, leading to unfeasible solutions. This results also holds true for star-shaped platforms.

Steady-state scheduling: In the second part, we discuss the issue of scheduling many copies of a given application, which is represented by a complex task graph. Instead of minimizing the completion time, we concentrate on the heart of the schedule and we try to maximize the throughput of the whole platform, without considering the start nor the end of our schedules. In this part, we first study the scheduling of complex but static applications, made of acyclic task graphs, on general heterogeneous platforms. To preserve a simple deployment of the application, produced schedules are made of a single allocation. Due to the NP-completeness of the problem, we not only provide an optimal solution, but also several heuristics returning efficient schedules. We compare our solutions to classical scheduling algorithms such as HEFT.
In a second step, we focus on a collection of simpler but dynamic applications to schedule on fully heterogeneous master-workers platforms: the characteristics of their instances are varying. Designing static schedules taking care of this dynamicity is difficult, even in case of simple bag-of-tasks applications. Assuming that these variations are represented by random variables, we provide an ε-approximation in clairvoyant context and efficient heuristics for both the semi-clairvoyant and non-clairvoyant cases. We present many simulations to assess their qualities compared to the Round-Robin or the On-Demand policies.
In a third step, we deal with pipeline applications, of which several tasks are replicated on different processors to increase the global throughput. In this case, even if instances are distributed in a simple Round-Robin fashion and if the mapping is completely specified, computing the throughput of the platform is difficult. We expose a model based on Timed Petri Nets to compute them; we also prove that the throughput can be computed in polynomial time for the Strict One-Port communication model.
Finally, steady-state techniques are effectively used to schedule complex task graph on a heterogeneous multi-core processor, the IBM Cell. We present a theoretical model of this processor and an efficient algorithm to schedule many instances of complex task graphs. An complete implementation of this algorithm shows strong performances, while actual throughputs are very close to those predicted by our solution.

Keywords:
Divisible loads, linear networks, multi-installlments, Steady-state scheduling, star-shaped platforms, master-worker, stochastic models, Timed Petri Nets, heuristics, linear programs, heterogeneous platforms, throughput maximization.