Chapter 8

Steady-State Scheduling

Olivier Beaumont
INRIA/LaBRI (UMR 5800, CNRS - Univ. Bordeaux 1 - ENSEIRB - Univ. Bordeaux 2)

Loris Marchal
CNRS/LIP (UMR 5668, CNRS - ENS Lyon - UCB Lyon 1 - INRIA)

8.1 Introduction 189
8.2 Problem formulation 191
8.3 Compact description of a schedule 193
8.4 From allocations and valid patterns to schedules 197
8.5 Problem solving in the general case 202
8.6 Toward compact linear programs 208
8.7 Conclusion 214

Abstract In this chapter, we propose a general framework for deriving efficient (polynomial-time) algorithms for steady-state scheduling. In the context of large scale platforms (grids or volunteer computing platforms), we show that the characteristics of the resources (volatility, heterogeneity) limit their use to large regular applications. Therefore, steady-state scheduling, that consists in optimizing the number of tasks that can be processed per time unit when the number of tasks becomes arbitrarily large, is a reasonable setting. In this chapter, we concentrate on bag-of-tasks and collective communications (broadcast and multicast) applications and we prove that efficient schedules can be derived in the context of steady-state scheduling, under realistic communication models that take into account both the heterogeneity of the resources and the contentions in the communication network.

8.1 Introduction

Modern computing platforms, such as Grids, are characterized by their large scale, their heterogeneity and the variations in the performance of their resources. These characteristics strongly influence the set of applications that can be executed using these platforms. First, the running time of the ap-

plication has to be large enough to benefit from the platform scale, and to minimize the influence of start-up times due to sophisticated middlewares. Second, an application executed of such a platform typically consists in many small tasks, mostly independent. This allows one to minimize the influence of variations in resource performance and to limit the impact of resource failures. From a scheduling point of view, the set of applications that can be efficiently executed on Grids is therefore restricted, and we can concentrate in this chapter on "embarrassingly parallel" applications consisting in many independent tasks. In this context, makespan minimization, i.e, minimizing the minimal time to process a given number of tasks, is usually intractable. It is thus more reasonable to focus on throughput maximization, i.e., to optimize the number of tasks that can be processed within T time units, when T becomes arbitrarily large.

Contrarily to the application model, the platform model may be rather sophisticated. Indeed, these platforms, made of the aggregation of many resources owned by different entities, are made of strongly heterogeneous computing resources. Similarly, due to long distance communications and huge volume of transmitted data, the cost of communications has to be explicitly taken into account. Some computation nodes within the same cluster may be able to communicate very quickly, whereas the communications between two nodes on both sides of the Atlantic may take much longer. Of course, predicting the exact duration of a 1GB communication through a transatlantic backbone is unreachable, but, as it is advocated in Chapter 11, the difficulties in estimating communication times should not lead us to neglect communications and assume a homogeneous network without congestion!

Therefore, when we consider scheduling issues on heterogeneous platforms, we need to cope with a rather complicated communication model (described in Section 8.2.1), but a rather simple application model (described in Section 8.2.2). Our goal in this chapter is to take advantage of the regularity of the applications; as we consider that the applications are made of a large number of identical operations, we relax the scheduling problem and consider the *steady-state* operation: we assume that after some transient initialization phase, the throughput of each resource will become stable. The idea behind Steady-State Scheduling is to relax the scheduling problem in order to only deal with resource activities and to avoid problems related to integral number of tasks. The question is therefore the following: Do scheduling problems become harder because we consider more sophisticated communication models or do they become simpler because we target simpler applications? In Section 8.3, we propose to represent a schedule by a weighted set of allocations (representing the way to transmit and execute a task) and by a set of weighted valid patterns (representing the way to organize communications and computations). In Section 8.4, we prove that it is possible to re-build a valid schedule from compatible sets of allocations and valid patterns. In Section 8.5, we prove that for many different applications under many different communication models, it is possible to find both the optimal solution (i.e.,

the optimal throughput) and to build a distributed schedule that achieves this throughput in strongly polynomial time, thus answering positively to the above question. At last, since the general framework proposed in Section 8.5 is based on the Ellipsoid method for solving linear programs and may lead to very expensive algorithms, we propose in Section 8.6 several efficient polynomial time algorithms for solving several Steady-State Scheduling problems.

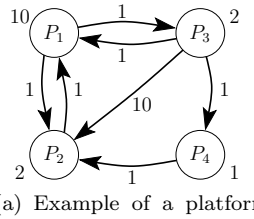
8.2 Problem formulation

In this section, we detail the modeling of the platform and of the target applications.

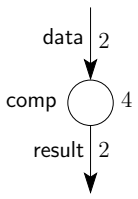
8.2.1 Platform model

A platform is represented by a *graph* $G = (V, E)$, where vertices correspond to processors and edges to communication links. For the sake of generality, we also introduce the concept of *resource*, that can either represent a processor (computation resource) or a link (communication resource). We denote by $\mathscr{R}$ the set of all resources.

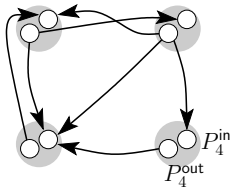
A processor P_u is characterized by its computing speed s_u, measured in flops (floating point operations per second). A communication link l_e is characterized by its bandwidth bw_e, measured in byte per second (B/s). For the sake of generality, we also extend the concept of *speed* to any resource r: s_r corresponds either to the computing speed if the resource is a processor, or to the bandwidth, if the resource is a communication link. Figure 8.1(a) gives an example of such a platform graph.



(a) Example of a platform graph. Processors and links are labelled with their speed. $P_{\mathsf{source}} = P_1$



(b) Example of bag-of-tasks application. Edges (files) and nodes (tasks) are labelled with their size.



(c) Bipartite communication graph for the bidirectional one-port model.

FIGURE 8.1: Example of platform graph and of bag-of-tasks applications.

Note that we do not take latencies into account: the communication of a message of size L on a link l_e takes L/bw_e time-units. The interaction between simultaneous communications are taken into account specifically by each model. In all cases, communications and computations can be overlapped.

Bidirectional one-port model: At a given time step, a processor can simultaneously be involved in two communications: sending a message and receiving another message.

Unidirectional one-port model: At a given time step, a processor can be involved in only one communication: either sending or receiving a message.

Bounded multi-port model: At given time step, a processor can be involved in several sending and receiving operations, provided that the total bandwidths used for incoming and outgoing communications does not exceed the capacity of the communication links. This model is close to the one proposed by Hong and Prasanna in [14].

We consider these different communication models as an illustration that steady-state scheduling can be used for a large range of models. However, in Section 8.6.2, we will more specifically study the bidirectional one-port model, which takes into account the fact that communications are usually serialized, and that in nowadays networks, most links are bidirectional (full-duplex).

8.2.2 Applications

Steady-state scheduling can be applied to a large range of applications, going from collective communications to structured applications. We present here three applications that will be detailed in the rest of this chapter.

The first application (and the simplest) is called a **bag-of-tasks** application. We consider an application consisting of a large number of independent and similar tasks. Although simple, this application models most of the embarrassingly parallel applications such as parameter sweep applications [9] or BOINC-like computations [8]. Each task of the bag consists in a message "data" containing the information needed to start the computation, a computational task denoted by "comp", and a message "result" containing the information produced by the computation. Initially, all data messages are hold by a given processor P_{source}. Similarly, all result messages must eventually be sent to P_{source}. All data messages have the same size (in bytes), denoted by δ_{data}, and all result messages have size δ_{result}. The cost of the each computational task comp (in flops) is w_{comp}. The simple application graph associated to the bag-of-tasks application is depicted in Figure 8.1(b). This application is close to the one presented in the chapter devoted to Divisible Load Scheduling (Chapter 7); however, even if, like in Divisible Load Schedul-

ing, we relax the problem to work with rational number of tasks, our ultimate goal is to build a schedule where task atomicity is preserved.

The second application is a collective communication operation between processors. It consists in broadcasting a message from one processor P_{source} to all other processors of the platform. No computation is induced by this operation, so that the only parameter to deal with is the size of the message data, δ_{data}. Broadcasting in computer networks is the subject of a wide literature, as parallel algorithms often require to send identical data to the whole computing platform, in order to disseminate global information (see [5] for extensive references). In steady-state scheduling, we concentrate on the **series of broadcast** problem: the source processor P_{source} has a large number of same-sized messages to broadcast to all other processors. We can symmetrically assume that the source processor has a message of large size to broadcast, which is divided into small chunks, and we target the pipelined scheduling of the series of chunks.

The third application considered here is the multicast operation, which is very close to the previous one: instead of broadcasting a message to all processors in the platform, we target only a given subset of the nodes denoted by $\mathcal{P}_{\mathsf{target}}$. As previously, we focus on the **series of multicast** problem, where a large number of same-sized messages has to be broadcast to target processors.

Just as we gather processors and communication links under the word "*resource*", we sometimes do not distinguish between computational tasks and files, and we call them "*operations*". Each operation is characterized by its size, which can either be the size of the corresponding message (in bytes) or the cost (in flops) of the corresponding computational task. The size of operation o is denoted by δ_o. Thus, the duration of operation o on resource r is δ_o/s_r. Note that this formulation forbids us to deal with *unrelated* computation models, where the execution time of a given task can vary arbitrarily among processors. It would be possible to study such complex computation models under the steady-state assumption, but at the cost of complicating the notations. Therefore, in this chapter, we limit our study to *related* computation models to keep notations simple.

At last, we denote by $\mathcal{O}$ the set of all operations in the considered application.

8.3 Compact description of a schedule

Scheduling an application on a parallel platform requires to describe *where* and *when* each task will be processed and each transfer will be executed. Since we concentrate on the problem of scheduling a large number of similar

jobs, this description may well have a very large size. Indeed, providing the explicit answer to above questions for each task would lead to a description of size $\Omega(n)$, thus pseudo-polynomial in the size of the input. Indeed, since all n tasks are identical, the size of the input is of order $\log n$ and not n. Therefore, we are looking for a compact (i.e., polynomial in the size of the input) description of the schedule.

In this section, we separate temporal and spatial description of such a schedule. We first explain how to get a compact description of the spatial aspect of the schedule (*where* operations are done); then we focus on the temporal aspect (*when* they are done).

8.3.1 Definition of the allocations

In order to introduce the concept of allocation, let us consider the bag-of-tasks application and the platform graph represented on Figure 8.1(a). We also assume that the source processor is P_1. The problem consists in scheduling a large number of similar tasks. We first concentrate on a *single task* in the series, and study *where*, i.e., on which processor this task can be processed and on which links the transfers of the data and result messages can be done.

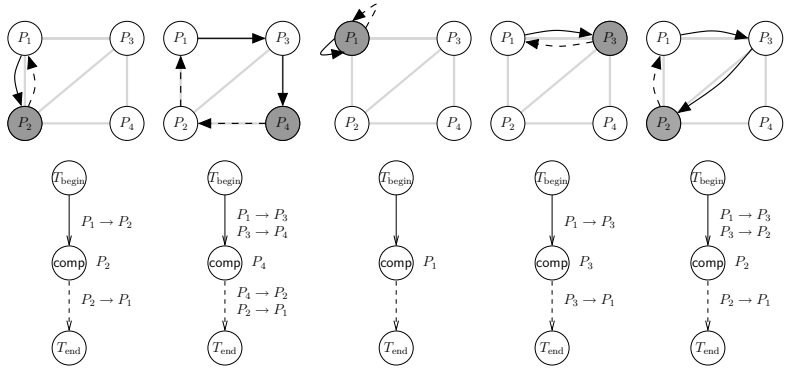
The computational task can be computed on any processor: P_1, P_2, P_3 or P_4. Once we have decided where this particular task will be processed, say P_2, we have to determine how the data (and the result) are sent from P_{source} to P_2 (respectively from P_2 to P_{source}). Any path may be taken to bring the data from the source to the computing processor: $P_1 \to P_2$, $P_1 \to P_3 \to P_2$ or $P_1 \to P_3 \to P_4 \to P_2$. To fully describe where a task is done, an allocation should provide (i) the path taken by the data, (ii) the processor computing the task, and (iii) the path taken by the result. Figure 8.2 presents some valid allocations for the bag-of-tasks application on the previous platform.

To cope with the different applications and to be able to schedule both communication primitives and complex scheduling problems, we propose the following general definition of an allocation.

DEFINITION 8.1 (allocation) *An allocation A is a function which associates a set of platform resources to each operation such that all constraints of the particular application are satisfied.*

To complete the definition for the three applications introduced above, we have to detail their respective constraints.

- For the **bag-of-tasks** application, the constraints on the allocation are the following:
  - The set of resources $A(\mathsf{comp})$ is a single processor;
  - The set of resource $A(\mathsf{data})$ is a path from processor P_{source} to $A(\mathsf{comp})$;



(a) A_1 (b) A_2 (c) A_3 (d) A_4 (e) A_5

FIGURE 8.2: Examples of allocations for a bag-of-tasks application.

  - The set of resource $A(\mathsf{result})$ is a path from $A(\mathsf{comp})$ to P_{source}.

- For the **series of broadcast** and the **series of multicast** applications, there is a single operation, which consists in the message data to be broadcast. The set of resources $A(\mathsf{data})$ must be a tree made of edges from the platform graph G, rooted at P_{source}. Furthermore, for the broadcast application, this tree has to span the whole platform graph, whereas for the multicast operation, the tree has to span a subset of the processors that contains all destination nodes.

Note that several resources may be associated to a single operation, like in the broadcast application (where several links are associated to the transfer of the data message). In this case, we assume that there is no need for a simultaneous use of these resources. For example, assume some legacy code used in a task imposes that four processors must be enrolled to process this task: such a task cannot be modeled with our framework, since we are unable to guarantee that the four processors allocated to the task will be available at the very same time for this particular task. On the contrary, in the case of the broadcast operation, each link of a given route can be used one after the other, by storing the message on intermediate nodes. Of course there is a precedence constraint (the links have to be used in a given order), and we are able to deal with this case in our model.

Allocations characterize where the computations and the transfers take place for a single task. Let us come back to the series of tasks problem: each task of the series has its own allocation. However, the number of allocations is finite: for instance, in the case of the bag-of-tasks application, there are at most $H^2 \times P$ different possible allocations, where P is the number of processors, and H the number of paths (without cycles) in the platform graph. We call $\mathscr{A}$ the set of possible allocations.

For each allocation $A \in \mathscr{A}$, we focus on the number of times this allocation is used, that is the number of tasks among the series that will be transfered and processed according to this allocation. Yet, as we do not want to depend on the total number of tasks in the series, we rather compute the *average throughput* of a given allocation: we denote by x_a the (fractional) number of times the allocation A_a is used per time-units (say, seconds).

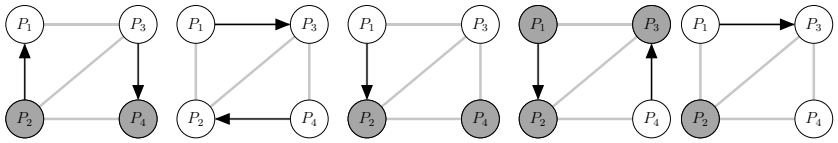
8.3.2 Definition of valid patterns

We have studied how to describe *where* the operations of a given application are executed. We now focus on the temporal aspect of the schedule, that is *when* all the operations involved by the allocations can take place. More specifically, we are looking for sets of operations (communications or computations) that can take place simultaneously. Such a set corresponds to a *valid pattern*.

DEFINITION 8.2 (valid pattern) *A valid pattern π is a set of operations (communications and/or computations) that can take place simultaneously according to a given model.*

We denote by Π the set of all possible valid patterns. According to the communication model we consider, a valid pattern corresponds to different structures in the platform graph.

Unidirectional one-port model. In this model, a processor can be involved in at most one communication, but computations and communications can be overlapped. The communication links involved in a simultaneous pattern of communications constitutes a matching in the platform graph. A valid pattern is therefore a matching in G plus any subset of computing processors. Some examples of valid patterns for this model are depicted in Figure 8.3.



(a) π_1 (b) π_2 (c) π_3 (d) π_4 (e) π_5

FIGURE 8.3: Examples of valid patterns for the unidirectional one-port model. Shaded processors and dark links are the resources taking part in the pattern.

Bidirectional one-port model. In this model, a processor can be involved in at most two communications, since it can simultaneously send and receive data. To better model this behavior, let us build the communication graph G_B obtained by splitting each processor P_u into two communicating processors: P_u^{out} is in charge of the sending messages while P_u^{in} is dedicated to receiving messages. All communication links are naturally translated into G_B: a link $P_u \to P_v$ is turned into a link $P_u^{\mathsf{out}} \to P_v^{\mathsf{in}}$ in G_B. G_B is a bipartite graph between P_*^{out} and P_*^{in} nodes, and the communications that can take place simultaneously in this model build up a matching in G_B. A valid pattern is thus made of a matching in G_B plus any subset of communicating processors. Figure 8.1(c) illustrates G_B for the platform graph described on Figure 8.1(a).

Bounded multi-port model. In this model, all operations (communications and computations) can take place simultaneously, provided that their aggregated throughput does not exceed platform capacity. Therefore, a valid pattern for this model is made of any subset of communication links and computing processors.

Since the computations are independent from the communications in all the models we consider, valid patterns are sometimes denoted "communication patterns" or even "matchings" when using the one-port communication models.

Each valid pattern may be used several times in a schedule. Rather than specifying for each task which pattern is used, we characterize each pattern π_p by its *average utilization time* y_p, that corresponds to the ratio between the time a given pattern is used and the total schedule time. We have considered an average throughput for the allocations; we similarly consider that a pattern π_p is used for a time y_p per time unit.

8.4 From allocations and valid patterns to schedules

8.4.1 Conditions and weak periodic schedules

In this section, we describe how to move from a description made of weighted allocations and weighted valid patterns to an actual schedule. Let us therefore consider a set of allocations with their average throughput, and a set of valid patterns with their average utilization time.

We first give necessary conditions on throughputs and utilization times to build a schedule based on these two sets. The first condition concerns resource activity. We consider a given resource r; this resource might be used by several tasks corresponding to different allocations. We further focus on a given allocation A_a; this allocation makes use of resource r for an operation o

if and only if $r \in A(o)$. Since operation o has size δ_o and resource r has speed s_r, r needs δ_o/s_r time units to perform operation o. Assuming that allocation A_a has an average throughput x_a, the total time needed on resource r for this allocation during one time-unit is given by

$$\sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r}.$$

We now focus on the time available on a given resource r. This resource is active and can be used during each valid pattern π_p where r appears. Assuming that the average utilization time of π_p is y_p, then the total availability time of resource r is given by

$$\sum_{\substack{\pi_p \in \Pi \\ r \in \pi_p}} y_p.$$

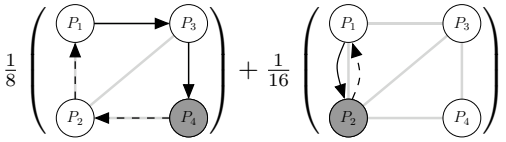
We are now able to write our first set of constraints: on each resource, the average time used by the allocations during one time-unit must be smaller than the average available time, i.e.,

$$\forall r \in \mathscr{R} \quad \sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r} \leqslant \sum_{\substack{\pi_p \in \Pi \\ r \in \pi_p}} y_p. \tag{8.1}$$

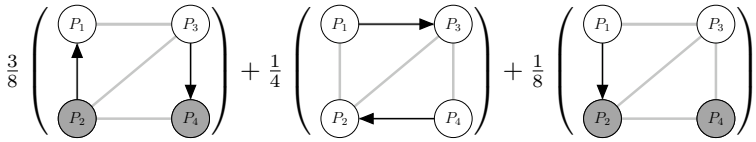
The second constraint comes from the definition of the availability times: y_p is the average time during which valid pattern π_p is used. During one time-unit, no more than one time-unit can be spent using all possible patterns, hence the second constraint

$$\sum_{\pi_p \in \Pi} y_p \leqslant 1. \tag{8.2}$$

These conditions must be satisfied by any pair of weighted sets of allocations and patterns that corresponds to a valid schedule. Figure 8.4 depicts an example of such sets, made of two allocations from Figure 8.2: A_1 with average throughput 1/8, and A_2 with throughput 1/16, and three valid patterns from Figure 8.3: π_1 with utilization time 3/8, π_2 with utilization time 1/4, and π_3 with utilization time 1/8. Let us consider for example the communication link $\pi_2 \to \pi_1$. It is used by both allocations to ship the result message of size 2, and has speed 1. The total utilization time of this communication link is therefore $\frac{1}{8} \times \frac{2}{1} + \frac{1}{16} \times \frac{2}{1} = \frac{3}{8}$. This link belongs to valid pattern π_1, with utilization time 3/8, thus Constraint (8.1) is satisfied on this link. We can similarly check that Constraint (8.1) is satisfied for each resource, and that the overall utilization time of valid patterns is given by $\frac{3}{8} + \frac{1}{4} + \frac{1}{8} \leqslant 1$, so that Constraint (8.2) is also satisfied.



(a) Allocations A_1 and A_2 with their throughputs.



(b) Valid patterns π_1, π_2, and π_3 with their utilization times.

FIGURE 8.4: Example of allocations and valid patterns satisfying conditions 8.1 and 8.2.

8.4.2 Weak periodic schedules and cyclic scheduling

In this section, we present the basic block of a steady-state schedule, which is called a weak periodic schedule. We also describe how to build such a weak periodic schedule based on allocations and valid patterns that satisfy previous conditions.

DEFINITION 8.3 (K-weak periodic schedule of length T) *A K-weak periodic schedule of length T is an allocation of K instances of the application within time T that satisfies resource constraints (but not the dependencies between operations constituting an instance).*

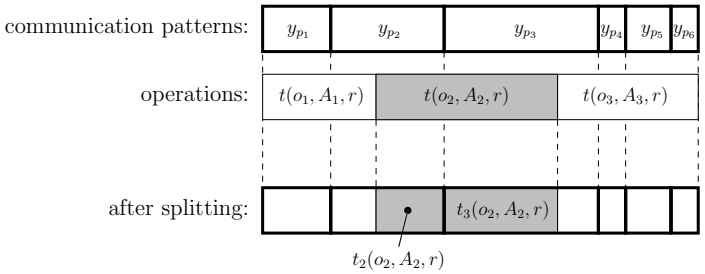
Here by *instance*, we mean a single task in the bag-of-tasks applications, or a single message to broadcast in the series of broadcast (or multicast). In a weak periodic schedule, we do not take precedence constraints into account: for example a data message does not have to be received before a computation comp takes place. The precedence constraints will be taken care of in the next section, and fulfilled by the use of several consecutive weak periodic schedules.

We present here a greedy algorithm to build a weak periodic schedule. We assume that we have a set of allocations weighted by their average throughput, and a set of valid patterns weighted by the average utilization time, that satisfy both conditions (8.1) and (8.2). Furthermore, we assume that the values of average throughputs and average utilization times are rational numbers.

1. For each resource r, we do the following:
   (a) We consider the set $\Pi(r)$ of valid patterns which contains resource r;
   (b) Let us consider the set L_r of pairs (o, A) such that $A(o)$ contains resource r; for each element (o, A) of this set, we compute the time

needed by this operation: $t(o, A, r) = x_a \frac{\delta_o}{s_r}$.
Thanks to Constraint (8.1), we know that

$$\sum_{(o,A)\in L_r} t(o, A, r) \leqslant \sum_{i=1}^{l} y_{p_i}$$



Then, we split each $t(o, A, r)$ into pieces so that they can fit into the intervals defined by the y_{p_i} (in any order), as illustrated above. We call $t_p(o, A, r)$ the time allocated for pair (o, A) using resource r on pattern π_p, and $n_p(o, A, r) = \frac{s_r}{\delta_o} \times t_p(o, A, r)$ the (fractional) number of instances of operation o for allocation A that can be computed in time $t_p(o, A, r)$.

2. We compute T, the least common multiple (lcm) of the denominators of all values $n_p(o, A, r)$, and we set $K = T \times \sum_{A_a \in \mathscr{A}} x_a$.

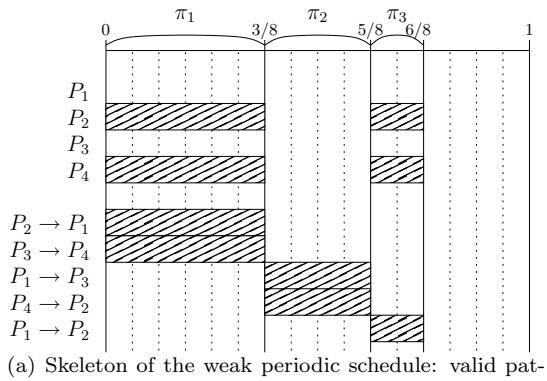
3. We build a K-weak periodic schedule of length T as follows. A weak periodic schedule is composed of $|\Pi|$ intervals of length $T \times y_p$. During the p-th interval, $\left[T \times \sum_{i=1}^{p-1} y_p \ ; \ \ T \times \sum_{i=1}^{p} y_p\right]$, the schedule "follows" the valid pattern π_p:

   - Resource r is active if and only if $r \in \pi_p$;
   - For each couple (o, A) such as $r \in A(o)$, r performs $T \times n_p(o, A, r)$ operations o for allocation A (in any order). Due to the definition of T, we know that $T \times n_p(o, A, r)$ is an integer.

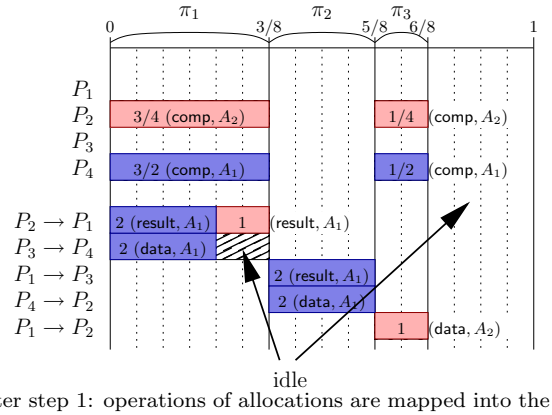
   Since Condition (8.2) is satisfied, the weak periodic schedule takes time $T \times \sum_{\pi_p \in \Pi} y_p \leqslant T$.

Figure 8.5 shows the construction of the weak periodic schedule corresponding to the allocations and valid patterns of Figure 8.4.

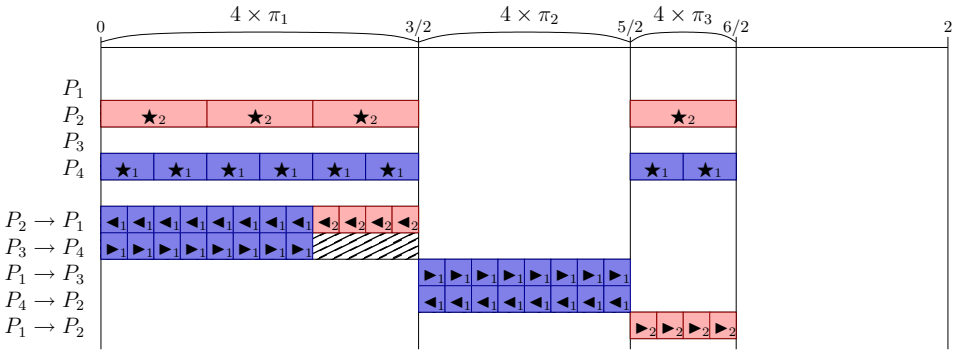
THEOREM 8.1
Given a K-weak periodic schedule of length T, we can build a cyclic schedule of period T, with throughput K/T.



(a) Skeleton of the weak periodic schedule: valid patterns with their utilization times.



(b) After step 1: operations of allocations are mapped into the slots of the patterns. Numbers represents quantities $n_p(o, A, r)$.



$\blacktriangleright_i$: transfer of data for allocation i
$\bigstar_i$: computation of comp for allocation i
$\blacktriangleleft_i$: transfer of result for allocation i

(c) Final weak periodic schedule, after scaling ($T = 4$), with all operations labelled.

FIGURE 8.5: Example of the construction of a weak periodic schedule from allocations and valid patterns of Figure 8.4

The throughput represents the average number of instances of the application that are performed per one time-unit in steady state. To prove this result and formally define the throughput of a cyclic schedule, important concepts of cyclic scheduling theory must be introduced, which is not the aim of this chapter. We refer the interested reader to [2] where the theory is detailed, and all previous theorems are proved for the problem of scheduling series of workflows (DAGs) on a heterogeneous platform.

We now have all the pieces to build a schedule from the allocations and valid patterns.

THEOREM 8.2
Given a set of allocations $\mathscr{A}$ weighted by their average throughput x and a set of valid patterns Π weighted by their average utilization time y, where x and y are vectors of rational values, if these sets satisfy Constraints (8.1) *and* (8.2)*, then we can build a periodic schedule with throughput $\sum_{A_a \in \mathscr{A}} x_a$.*

This result is a simple corollary of Theorem 8.1 applied to the weak periodic schedule builded from the sets of allocations and valid patterns.

8.5 Problem solving in the general case

In this section, we focus on the construction of allocations and valid patterns satisfying Constraints (8.1) and (8.2) and that maximize the overall throughput. We first gather this objective together with the constraints into a linear program:

$$\text{MAXIMIZE } \rho = \sum_{A_a \in \mathscr{A}} x_a, \text{ UNDER THE CONTRAINTS}$$
$$\begin{cases} \displaystyle\sum_{\pi_p \in \Pi} y_p \leqslant 1, \\ \forall r \in \mathscr{R} \quad \displaystyle\sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r} \leqslant \sum_{\substack{\pi_p \in \Pi \\ r \in \pi_p}} y_p, \\ \forall A_a \in \mathscr{A}, \quad x_a \geqslant 0, \\ \forall \pi_p \in \Pi, \quad y_p \geqslant 0. \end{cases} \tag{8.3}$$

This linear program is not directly tractable. Indeed, both the number of possible allocations and the number of possible valid patterns can be of exponential size in the size of the original problem. Therefore, the number of variables of this linear program may be huge. To simplify the notations, let

us rewrite the previous linear program as follows

$$\text{MAXIMIZE } c^T \cdot X, \text{ UNDER THE CONTRAINTS}$$

$$\begin{cases} A \cdot X \leqslant b, \\ X \geqslant 0. \end{cases} \tag{P}$$

Let us denote by $m = |\mathscr{A}| + |\Pi|$ the number of variables and by $n = |\mathscr{R}| + 1$ the number of non-trivial constraints, then matrix A has size $n \times m$, b is a vector of size n, and c a vector of size m. Variable X gathers both the x_a and the y_p: $X_i = x_i$ for $1 \leqslant i \leqslant |\mathscr{A}|$ and $X_{|\mathscr{A}|+i} = y_i$ for $1 \leqslant i \leqslant |\Pi|$. Matrix A is described below:

	for each allocation A_a	for each valid pattern π_p
	0,0,...,0	1,1,...,1
for each resource $r \in \mathscr{R}$	$\sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} \frac{\delta_o}{s_r}$	(-1 if $r \in \pi_p$, 0 otherwise)

8.5.1 Existence of a compact solution

A solution of the above linear program is *a priori* described by a huge number of weighted allocations and a huge number of valid patterns. In this section, we prove that there exists a compact optimal solution, i.e., an optimal solution with polynomial size in the size of the problem.

We write each δ_o and each s_r as irreducible fractions $\frac{\alpha_o}{\beta_o}$ and $\frac{\alpha'_r}{\beta'_r}$, and we compute $\Delta = \max\{\alpha_o, \beta_o, \alpha'_r, \beta'_r\}$. Thus, the encoding of each coefficient of matrix A has size at most $2 \log \Delta$.

The number of constraints in the linear program is not polynomial in the size of the problem. However, most of these constraints are trivial ($X \geqslant 0$), and only $|\mathscr{R}| + 1$ constraints need to be checked through computations. Thus, we assume that we have an oracle which, given a solution $(i_1, X_{i_1} = a_{i_1}/b_{i_1}), \ldots, (i_k, X_{i_k} = a_{i_k}/b_{i_k})$ (we implicitly assume that $X_i = 0$ on all other components) and for all $K \in \mathbb{Q}$, can determine if the following system of constraints is satisfied in time $O(k^\gamma \log(\max\{a_{i_j}, b_{i_j}\})$:

$$\begin{cases} A \cdot X \leqslant b, \\ X \geqslant 0, \\ c^T \cdot X \geqslant K. \end{cases}$$

The last constraint enables to check if the throughput of the solution is above a given threshold K, as in the decision problem associated to the previous linear program.

DEFINITION 8.4 LIN-PROG-DEC *Does there exists $X \in \mathbb{Q}^m$ which satisfies the constraints of Linear Program* (P) *and such that the value of the objective function verifies $c^T \cdot X \geqslant K$?*

Since the number of constraints to be checked is huge, this problem does not a priori belongs to the $\mathcal{NP}$ class. We consider the following reduced decision problem.

DEFINITION 8.5 RED-LIN-PROG-DEC *Does there exist $X \in \mathbb{Q}^m$ such that:*

(i) X has at most n positive components;

(ii) the positive components of X, $X_i = a_i/b_i$, satisfy:

$$\log(a_i) + \log(b_i) \leqslant B$$

(iii) X satisfies the constraints of Linear Program (P)*, and $c^T \cdot X \geqslant K$?*

The complexity bound B depends on Δ, and is set to $B = 2n(\log n + 2n \log \Delta) + n \log \Delta$ for technical reasons. We are able to prove that these two versions of the decision problem are equivalent, but the second one belongs to $\mathcal{NP}$.

THEOREM 8.3
RED-LIN-PROG-DEC belongs to the $\mathcal{NP}$ class, and if there exists a solution X of LIN-PROG-DEC, then there exists a solution Y of RED-LIN-PROG-DEC, and Y is also solution of LIN-PROG-DEC.

PROOF We briefly present the idea of the proof. The detailed proof can be found in [18].

It is easy to check that RED-LIN-PROG-DEC belongs to $\mathcal{NP}$: given a solution that satisfies the conditions of the decision problem, we can apply the oracle desribed above to this solution, and check that it is a valid solution in polynomial time.

To prove the second part of the theorem, we consider a solution X of LIN-PROG-DEC. We know that there exists an optimal solution Y of the linear program that corresponds to a vertex of the convex polyhedron defined by the constraints of (P). Y can be obtained by solving a linear system of size $m \times m$ extracted from the constraints of (P). Since A has size $n \times m$, the linear systems contains at most n rows from matrix A, and at least $m - n$ rows $X_i = 0$. This means that at most n components of Y are different from zero (point (i)).

To prove that the positive components of Y have a bounded complexity (point (ii) of RED-LIN-PROG-DEC), we can go further on the computation of Y with the linear system, and consider that each component of Y is obtained

with Cramer formulae [11] as a fraction of two determinants extracted from A. This allows to bound the value of the numerator and denominator of all elements (point (iii)).

Finally, since Y is an optimal solution, its throughput is larger that the throughput of X. □

This proves that our expression of the problem makes sense: there exist compact, easily described optimal solutions, and we now aim at finding them.

8.5.2 Resolution with the Ellipsoid method

We now focus on throughput optimization. The method we propose is based on the Ellipsoid method for linear programming, introduced by Khachiyan [16] and detailed in [19]. To apply this method, we consider the dual linear program of (P)

$$\text{Minimize } b^T U, \text{ under the contraints}$$

$$\begin{cases} A^T \cdot U \geqslant c, \\ U \geqslant 0. \end{cases} \tag{D}$$

Let us analyze this linear program. There is one constraint per allocation A_a, and one per valid pattern π_p, whereas there is one variable (U_1) corresponding to Constraint (8.1) and one variable (U_{r+1}) for each constraint of type (8.2), i.e., one for each resource r. The previous linear program can also be written:

$$\text{Minimize } b^T U, \text{ under the contraints}$$

$$\begin{cases} \textit{(I)} \quad \forall A_a \in \mathscr{A} & \displaystyle\sum_{r \in \mathscr{R}} \sum_{\substack{o \in \mathscr{O} \\ r \in A \ a(o)}} \frac{\delta_o}{s_r} U_{r+1} \geqslant 1, \\ \textit{(II)} \quad \forall \pi_p \in \Pi & \displaystyle\sum_{r \in \pi_p} U_{r+1} \leqslant U_1, \\ \textit{(III)} & U \geqslant 0. \end{cases} \tag{D_2}$$

Given an optimization problem on a convex and compact set K of $\mathbb{Q}^k$, we consider the following two problems. The first one is the optimization problem.

***DEFINITION 8.6 OPT**(K, v)* *Given a convex and compact set K and a vector $v \in \mathbb{Q}^k$, find a vector $x \in K$ which maximizes $v^T \cdot x$, or prove that K is empty.*

The second problem is a separation problem, which consists in deciding whether the convex K contains a vector v and, if not, in finding an hyperplane separating v from K.

DEFINITION 8.7 SEP(K, v) *Given a convex and compact set K and a vector $v \in \mathbb{Q}^k$, decide if $v \in K$ and if not, find a vector x such that $x^T \cdot v > \max\{x^T \cdot y,\ y \in K\}$.*

The Ellipsoid method is based on the equivalence between above two problems, as expressed by the following result [12, Chapter 6].

THEOREM 8.4
Each of the two problems, OPT(K, v) and SEP(K, v) can be solved in polynomial time for each "well described" polyhedron if we know a polynomial-time oracle for the other problem.

A convex polyhedron is "well described" if it can be encoded in polynomial size, which is possible in our case (see [18] for details).

THEOREM 8.5 (Theorem 6.5.15 in [12])
There exists a polynomial-time algorithm, which, for $x \in \mathbb{Q}^k$ and a well described polyhedron (corresponding to the dual (D)*) described by a polynomial-time separation oracle,*
(i) finds a solution to the primal problem (P)
(i) or proves that (P) *has unbounded solutions or no solution at all.*

To solve the optimization problem using Ellipsoid method, it is therefore enough to find a polynomial-time algorithm that solves the separation problem in (D) (or (D_2)).

8.5.3 Separation in the dual linear program

Given a vector $U \in \mathbb{Q}^k$ (with $k = |\mathscr{R}| + 1$), we can check whether U satisfies all the constraints of (D_2), and if not, find a constraint which is violated. This unsatisfied constraint provides a hyperplane separating vector U from the convex. There are three types of constraints in the linear program. The last constraints (type *(III)*) are easy to check, as the number of variables is small. The other constraints can be linked with allocations (type *(I)*) or valid patterns (type *(II)*).

8.5.3.1 Allocation constraints

Let us first consider constraints of type *(I)*. To each resource r (processor or link), we associate a weight $w_r = \sum_{o \in \mathcal{O}} \frac{\delta_o}{s_r} U_{r+1}$. The constraint for allocation A_a is satisfied if the weighted sum of the resources taking part in the allocation is larger or equal to one. We specify this for the three examples of applications:

Broadcast. An allocation is a spanning tree in the platform graph, rooted at P_{source}. The constraint for allocation A_a states that the weight of the tree is larger than one. In order to check all constraints, we compute (in polynomial time) a spanning tree of minimum weight $A_{\min}$ [10]. If its weight is larger or equal to one, then all spanning trees satisfy this constraint. On the contrary, if its weight is smaller than one, then the corresponding constraint is violated for this minimal weight tree.

Multicast. The allocations for the multicast operation are very closed to the ones of the broadcast: an allocation is a tree rooted at P_{source} but spanning only the nodes participating to the multicast operation. Once the platform graph is weighted as above, checking all the constraints can be done by searching for a minimum weight multicast spanning tree. However, finding a minimum tree spanning a given subset of the nodes in a graph is a well known NP-complete problem, also called the Steiner problem [15]. This suggests that the steady-state multicast problem is difficult, but does not provide any proof. Nevertheless, finding the optimal throughput for the series of multicast problem happens to be NP-complete [4].

Bag-of-tasks application. For this problem, the allocations are a little more complex than simple trees, but finding the allocation with smallest weight can still be done in polynomial time: each allocation consists in a computing processor P_{comp}, a path from the source to the computing processor $P_{\mathsf{source}} \rightsquigarrow P_{\mathsf{comp}}$, and a path back $P_{\mathsf{comp}} \rightsquigarrow P_{\mathsf{source}}$. Note that the choices of the two paths are made independently and they may have some edges in common: an edge (and its weight) is thus counted twice if it is used by both paths. A simple method to compute the allocation of minimum weight is to consider iteratively each processor: for each processor P_u, we can find in polynomial time a path of minimum weight from P_{source} to P_u [10], and another minimum weight path from P_u to P_{source}. By taking the minimum of the total weight among all computing processors, we obtain a minimum weight allocation.

8.5.3.2 Valid pattern constraints

The constraints of type *(II)* are quite similar of the constraints of type *(I)*: for each valid pattern, we have to check that the sum of the weights of the resources involved in this pattern is smaller than a given threshold U_1, where the weight of a resource r is U_{r+1}. We proceed very similarly to the pre-

vious case: we search a valid pattern with maximum weight, and check if it satisfies the constraint: if it does, all other valid patterns (with smaller weights) lead to satisfied constraints, and if it does not, it constitutes a violated constraint. Finding a pattern with maximum weight depends on the communication model in use.

Unidirectional one-port model. A valid pattern in this model corresponds to a matching in the communication graph plus any subset of computing nodes. To maximize its weight, all computing processors can be chosen, and the problem turns into finding a maximum weighted matching in the graph, what can be done in polynomial time [10].

Bidirectional one-port model. In this model, a valid pattern is a matching in the bipartite communication graph G_B, plus any subset of computing nodes. Again, the problems turns into finding a maximum weighted matching (in a bipartite graph in this context).

Bounded multi-port model. In that case, all communications and computations can be done simultaneously. A valid pattern with maximum weight is thus the one including all processors and all communication links.

Thus, we can solve the separation problem in the dual (D_2) in polynomial time. Using Theorem 8.5, we can solve the original linear program (8.3), i.e.,

- Compute the optimal steady-state throughput;
- Find weighted sets of allocations and valid patterns to reach this throughput;
- Construct a periodic schedule realizing this throughput, with the help of Theorem 8.2.

The framework we have developed here is very general. It can thus be applied to a large number of scheduling problems, under a large number of communication models, as soon as we can explicit the allocations and the valid patterns, and solve the separation problem for these allocations and patterns. This allows us to prove that many scheduling problems have a polynomial complexity when studied in the context of steady-state scheduling.

8.6 Toward compact linear programs

8.6.1 Introduction

In previous sections, we have proposed a general framework to analyze the complexity of steady-state scheduling problems. This framework is based on the polynomial-time resolution of linear programs (see Linear Program (8.3)). The variables in (8.3) are the set of all possible allocations and valid patterns and thus may both be of exponential size in the size of the platform. Therefore, the Ellipsoid method is the only possible mean to solve these lin-

ear programs in polynomial time, since all other methods require to explicitly write the linear program. We have proved that this method enables the design of polynomial-time algorithms for a large number of problems under a large number of communication schemes. Nevertheless, the Ellipsoid method is known to have a prohibitive computational cost, so that it cannot be used in practice.

Our goal in this section is to provide efficient polynomial-time algorithms for several steady-state scheduling problems. More precisely, we prove in Section 8.6.2, that the set of variables corresponding to valid patterns can be replaced by a much smaller set of variables and constraints dealing with local congestion only. In Section 8.6.3, we show that for a large set of problems, the set of variables dealing with allocations can also be replaced by a more compact set of variables and constraints. In this latter case, optimal steady-state schedules can be efficiently solved in practice, since corresponding linear programs only involve polynomial number of variables and constraints (with low degree polynomials). Nevertheless, for some problems, especially those under the unidirectional one-port model, no efficient polynomial algorithm is known, whereas those problems lie in $\mathcal{P}$, as assessed in the previous section.

8.6.2 Efficient computation of valid patterns under bidirectional one-port model

Steady-state scheduling problems can be formulated as linear programs where there is one variable per allocation and per valid pattern (see Linear Program (8.3)). In the case of bidirectional one port model, the variables corresponding to valid patterns can be replaced by variables and constraints dealing with local congestion only. Indeed, let us denote by $t_{u,v}$ the occupation time of the communication link from P_u to P_v induced by the allocations. Then,

$$\forall r = (P_u, P_v) \in \mathscr{R} \quad t_{u,v} = \sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r}.$$

In the bidirectional one-port model, incoming and outgoing communications at a node P_u can take place simultaneously, but all incoming (respectively outgoing) communications at P_u must be sequentialized. Therefore, we can define t_u^{in} (resp. t_u^{out}), the time when P_u is busy receiving (resp. sending) messages as

$$t_u^{\mathsf{in}} = \sum_{(P_v,P_u)\in E} t_{v,u} \quad \text{and} \quad t_u^{\mathsf{out}} = \sum_{(P_u,P_v)\in E} t_{u,v},$$

and both quantities must satisfy $t_u^{\mathsf{in}} \leqslant 1$ and $t_u^{\mathsf{out}} \leqslant 1$. Therefore, when considering only constraints on communications, we can replace the original linear program by the following one,

$$\text{MAXIMIZE } \rho = \sum_{A_a \in \mathscr{A}} x_a, \text{ UNDER THE CONTRAINTS}$$

$$\begin{cases} \forall r = (P_u, P_v) \in \mathscr{R} & t_{u,v} = \displaystyle\sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r}, \\ \forall r = P_u \in \mathscr{R} & t_u^{\mathsf{in}} = \displaystyle\sum_{(P_v, P_u) \in E} t_{v,u}, \quad t_u^{\mathsf{in}} \leqslant 1, \\ & t_u^{\mathsf{out}} = \displaystyle\sum_{(P_u, P_v) \in E} t_{u,v}, \quad t_u^{\mathsf{out}} \leqslant 1, \end{cases} \tag{8.4}$$

where the set of variables dealing with valid patterns, which had an exponential size *a priori*, have been replaced by $|E| + 2|V|$ variables and constraints. The following theorem states that this modification in the linear program does not affect the optimal throughput.

THEOREM 8.6
On an application using only communications (and not computations), both Linear Programs (8.3) *and* (8.4) *provide the same optimal objective value.*

PROOF Let us first consider a solution of Linear Program (8.3). Then,

$$\forall r = (P_u, P_v) \in \mathscr{R}, \quad t_{u,v} = \sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r} \leqslant \sum_{\substack{P = \pi_p \in \Pi \\ (P_u, P_v) \in \pi_p}} y_p$$

and therefore

$$t_u^{\mathsf{out}} = \sum_{(P_u, P_v) \in E} t_{u,v} \leqslant \sum_{(P_u, P_v) \in E} \sum_{\substack{\pi_p \in \Pi \\ (P_u, P_v) \in \pi_p}} y_p.$$

Moreover, the edges (P_u, P_v) and (P_u, P_k) cannot appear simultaneously in a valid pattern π_p, so that each π_p appears at most once in the above formula. Therefore,

$$t_u^{\mathsf{out}} \leqslant \sum_{\pi_p \in \Pi} y_p \leqslant 1.$$

Similarly, $t_i^{\mathsf{in}} \leqslant 1$ holds true and the t_u's satisfy the constraints of Linear Program (8.4).

On the other hand, let us consider an optimal solution of Linear Program (8.4) and let us build the bipartite graph G_B (see Section 8.3.2) representing communications, where the weight of the edge between the outgoing port of P_u (denoted by P_u^{out}) and the incoming port of P_v (denoted by P_v^{in}) is given by $t_{u,v}$. This bipartite graph can be decomposed into a set of matchings,

using the refined version of the Edge Coloring Lemma [19, vol. A, Chapter 20]:

Corollary 20.1a in [19, vol. A, Chapter 20]. *Let $G_B = (V', E', t_{u,v})$ be a weighted bipartite graph. There exist K matchings $M_1, \ldots, M_K$ with weights $\mu_1, \ldots, \mu_K$ such that*

$$\forall u, v, \quad \sum_{i=1}^{K} \mu_i \chi_{u,v}(M_i) = t_{u,v},$$

where $\chi_{u,v}(M_i) = 1$ if $(P_u^{\mathsf{out}}, P_v^{\mathsf{in}}) \in M_i$ and 0 otherwise, and

$$\sum_{i=1}^{K} \mu_i = \max\left(\max_u \sum_v t_{u,v}, \max_v \sum_u t_{u,v} \right).$$

Moreover, the matchings can be found in strongly polynomial time and by construction $K \leqslant |E'|$.

We build valid patterns directly from the matchings M: for M_i, if $\chi_{u,v}(M_i) = 1$, then we include the communication link $(P_u^{\mathsf{out}}, P_v^{\mathsf{in}})$ in the pattern π_i. Therefore, we can build from the solution of Linear Program (8.4) a set of valid patterns (the matchings M_i) to organize the communications. Thus, both linear programs provide the same optimal objective value. □

For the sake of simplicity, we have not considered processing resources in this section. To take processing into account, we can bound the occupation time of a processor with a new constraint in the linear program:

$$\forall r = P_u \in \mathscr{R} \quad \sum_{A_a \in \mathscr{A}} \sum_{\substack{o \in \mathscr{O} \\ r \in A(o)}} x_a \frac{\delta_o}{s_r} \leqslant 1. \tag{8.5}$$

Then, since communications and computations do not interfere, we can schedule the computations of processor P_u, which last $t_u = \sum_o x_a \delta_o / s_u$, during the first t_u time units to build the valid patterns.

Therefore, in the case of the bidirectional one-port model, it is possible to replace the (exponential size) set of variables representing valid patterns by a much smaller (polynomial size) set of variables and constraints dealing with local congestion. This holds also true in the case of the bounded multi-port model, since building the valid patterns from local transfers is also equivalent to pealing a graph into a sum of weighted edge subsets. On the other hand, in the case of the unidirectional one-port model, the corresponding graph is not bipartite and it is therefore not possible to express the maximal throughput via local congestion constraints only. In fact, the design of efficient polynomial time algorithms for optimizing steady-state throughput is still completely open (for bag-of-tasks application, broadcast, ...).

8.6.3 Efficient computation of allocations

In the previous section, we have shown how the set of variables dealing with valid patterns can be replaced by a much smaller set of variables and constraints in the case of the bidirectional one-port (and bounded multi-port) model. However, an exponential set of variables still remains, describing the allocations.

In this section, we will concentrate on the bidirectional one-port model and prove how the (exponential size) set of variables dealing with allocations can also be replaced by a polynomial size set of variables and constraints. For simplicity, we will focus only on independent task distribution, but the same framework also applies to broadcasting [5] and independent task graphs scheduling [6] (under some conditions). The approach we propose is closely related to the pioneering work of Bertsimas and Gamarnik [7].

Let us consider the **bag-of-tasks** application problem described in Section 8.2.2, where a source (master) node P_{source} holds a large number of identical independent tasks to be processed by distant nodes (workers). All data messages have the same size (in bytes), denoted by δ_{data}, and all result messages have common size δ_{result}. The cost of any computational task comp is w_{comp} (in flops). We denote by s_u the computational speed of P_u and by $s_{u,v}$ the speed of the link between P_u and P_v.

Let us first concentrate on worker P_u and let us derive a set of equations corresponding to the communications and the processing induced by the execution of α_u tasks per time unit on P_u. First, P_u should be able to process the tasks, i.e., $\frac{\alpha_u \times w_{\mathsf{comp}}}{s_u} \leqslant 1$. Let us now introduce the new variables $\alpha^{i,j}_{u,\mathsf{data}}$ (respectively $\alpha^{i,j}_{u,\mathsf{result}}$) that represent the number of data (resp. result) messages for the tasks processed by P_u which are transmitted through link (P_i, P_j). Clearly, all data (resp. result) messages should leave (resp. reach) P_{source} and reach (resp. leave) P_u and therefore,

$$\sum_j \alpha^{\mathsf{source},j}_{u,\mathsf{data}} = \sum_i \alpha^{i,u}_{u,\mathsf{data}} = \alpha_u = \sum_j \alpha^{u,j}_{u,\mathsf{result}} = \sum_i \alpha^{i,\mathsf{source}}_{u,\mathsf{result}}$$

and no messages should be created or lost at any node different from P_{source} and P_u

$$\forall P_i,\ P_i \neq P_{\mathsf{source}},\ P_i \neq P_u, \quad \begin{cases} \displaystyle\sum_j \alpha^{i,j}_{u,\mathsf{data}} = \sum_j \alpha^{j,i}_{u,\mathsf{data}}, \\ \displaystyle\sum_j \alpha^{i,j}_{u,\mathsf{result}} = \sum_j \alpha^{j,i}_{u,\mathsf{result}}. \end{cases}$$

Let us now consider the transfers corresponding to all messages to all workers simultaneously. P_i is busy receiving data and result messages during

$$t_i^{\mathsf{in}} = \sum_u \sum_j \frac{\alpha^{j,i}_{u,\mathsf{data}} \times \delta_{\mathsf{data}}}{s_{j,i}} + \sum_u \sum_j \frac{\alpha^{j,i}_{u,\mathsf{result}} \times \delta_{\mathsf{result}}}{s_{j,i}}$$

and P_i is busy sending data and result messages during

$$t_i^{\mathsf{out}} = \sum_u \sum_j \frac{\alpha_{u,\mathsf{data}}^{i,j} \times \delta_{\mathsf{data}}}{s_{i,j}} + \sum_u \sum_j \frac{\alpha_{u,\mathsf{result}}^{i,j} \times \delta_{\mathsf{result}}}{s_{i,j}}.$$

Therefore, the following linear program provides an upper bound on the possible number of tasks that can be processed, at steady state, during one time-unit

$$\text{MAXIMIZE } \rho = \sum_u \alpha_u, \text{ UNDER THE CONTRAINTS}$$

$$\begin{cases} \forall P_u, \quad \alpha_u \geqslant 0, \quad \dfrac{\alpha_u \; w_{\mathsf{comp}}}{s_u} \leqslant 1, \\ \forall P_u, \quad \displaystyle\sum_j \alpha_{u,\mathsf{data}}^{\mathsf{source},j} = \sum_j \alpha_{u,\mathsf{data}}^{j,u} = \alpha_u = \sum_j \alpha_{u,\mathsf{result}}^{u,j} = \sum_j \alpha_{u,\mathsf{result}}^{j,\mathsf{source}} , \\ \forall P_i, \; P_i \neq P_{\mathsf{source}}, \; P_i \neq P_u, \quad \begin{cases} \displaystyle\sum_j \alpha_{u,\mathsf{data}}^{i,j} = \sum_j \alpha_{u,\mathsf{data}}^{j,i} , \\ \displaystyle\sum_j \alpha_{u,\mathsf{result}}^{i,j} = \sum_j \alpha_{u,\mathsf{result}}^{j,i} , \end{cases} \\ t_i^{\mathsf{in}} = \displaystyle\sum_u \sum_j \frac{\alpha_{u,\mathsf{data}}^{j,i} \times \delta_{\mathsf{data}}}{s_{j,i}} + \sum_u \sum_j \frac{\alpha_{u,\mathsf{result}}^{j,i} \times \delta_{\mathsf{result}}}{s_{j,i}} , \\ t_i^{\mathsf{out}} = \displaystyle\sum_u \sum_j \frac{\alpha_{u,\mathsf{data}}^{i,j} \times \delta_{\mathsf{data}}}{s_{i,j}} + \sum_u \sum_j \frac{\alpha_{u,\mathsf{result}}^{i,j} \times \delta_{\mathsf{result}}}{s_{i,j}} , \\ \forall P_i, \quad t_i^{\mathsf{in}} \leqslant 1, \quad t_i^{\mathsf{out}} \leqslant 1. \end{cases} \tag{8.6}$$

From the solution of the linear program, the set of valid patterns can be determined using the general framework described in Section 8.6.2. In order to build the set of allocations, we can observe that the set of values $\alpha_{u,\mathsf{data}}^{i,j}$ (resp. $\alpha_{u,\mathsf{result}}^{i,j}$) defines a flow of value α_u between P_{source} and P_u (resp. P_u and P_{source}). Each of these flows can be decomposed into a weighted set of at most $|E|$ disjoint paths [19, vol. A, Chapter 10]. Thus, it is possible to find at most $2|E|$ weighted allocations that represent the transfer of data from P_{source} to P_u, the processing on P_u and the transfer of results from P_u to P_{source}. Therefore, under the bidirectional one-port model, the Linear Program (8.6) provides the optimal throughput for the **bag-of-tasks** application problem. Moreover, this linear program is compact since it only involves $\Theta(|V||E|)$ variables and constraints.

8.7 Conclusion

Figure 8.7 summarizes the complexity of the problem from the point of view of steady-state scheduling. All NP-complete problems of this table are NP-complete for any of the communication models we have considered. For the problems with polynomial complexity, we do not know a better algorithm than the one using the Ellipsoid method (see Section 8.5.2) for the unidirectional one-port model, whereas we have efficient algorithms under the bidirectional one-port model (as for the bag-of-tasks application in Section 8.6.3).

	NP-complete problems	**problems with polynomial complexity**
collective operations	multicast and prefix computation [4]	broadcast [5], scatter and reduce [17]
scheduling problems	general DAGs [3]	bags of tasks, DAGs with bounded dependency depth [3]

Table 8.1: Complexity results for steady-state problems.

In this chapter, we have shown that changing the metric, from makespan minimization to throughput maximization, enables to derive efficient polynomial time algorithms for a large number of scheduling problems involving heterogeneous resources, even under realistic communication models where contentions are explicitly taken into account. Nevertheless, from an algorithmic point of view, large scale platforms are characterized by their heterogeneity and by the dynamism of their components (due to processor or link failures, workload variation,...). Deriving efficient scheduling algorithms that can self-adapt to variations in platforms characteristics is still an open problem. We strongly believe that in this context, one needs to inject some static knowledge into scheduling algorithms, since greedy algorithms are known to exhibit arbitrarily bad performance in presence of heterogeneous resources. For instance, Yu-Tong He et al. propose in [13] to base dynamic mapping decisions in resource management systems on the solution of a linear program, and Awerbuch and Leighton [1] show how to derive fully distributed approximation algorithms for multi-commodity flow problems based on potential queues whose characteristics are given by the pre-computation of the optimal solution.

References

[1] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *26-th annual ACM symposium on Theory of computing*, pages 487–496, 1994.

[2] O. Beaumont. *Nouvelles méthodes pour l'ordonnancement sur plates-formes hétérogènes*. Habilitation à diriger des recherches, Université de Bordeaux 1, Dec. 2004.

[3] 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. Reserach report 2004-20, LIP, ENS Lyon, France, Apr. 2004. Also available as INRIA Research Report RR-5198.

[4] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Complexity results and heuristics for pipelined multicast operations on heterogeneous platforms. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP)*. IEEE Computer Society Press, 2004.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Pipelining broadcasts on heterogeneous platforms. *IEEE Transactions on Parallel Distributed Systems*, 16(4):300–313, 2005.

[6] 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.

[7] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[8] Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu`.

[9] H. Casanova and F. Berman. *Grid Computing: Making The Global Infrastructure a Reality*, chapter Parameter Sweeps on the Grid with APST. John Wiley, 2003. Hey, A. and Berman, F. and Fox, G., editors.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] G. H. Golub and C. F. V. Loan. *Matrix computations*. Johns Hopkins, 1989.

[12] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithm and Combinatorial Optimization*. Algorithms and Combinatorics 2. Springer-Verlag, 1994. Second corrected edition.

[13] Y.-T. He, I. Al-Azzoni, and D. Down. MARO-MinDrift affinity routing for resource management in heterogeneous computing systems. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 71–85, New York, NY, USA, 2007. ACM.

[14] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[15] R. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pages 85–103. Plenum, New York, 1972.

[16] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademiia Nauk SSSR*, 224:1093–1096, 1979. English Translation: *Soviet Mathematics Doklady*, Volume 20, pp. 191–194.

[17] A. Legrand, L. Marchal, and Y. Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 65(12):1497–1514, 2005.

[18] L. Marchal. *Communications collectives et ordonnancement en régime permanent sur plates-formes hétérogènes*. Thèse de doctorat, École Normale Supérieure de Lyon, France, Oct. 2006.

[19] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.