Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms

Olivier Beaumont,
Arnaud Legrand,
Loris Marchal,
Yves Robert

April 2004

Research Report N° 2004-20

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms

Olivier Beaumont, Arnaud Legrand, Loris Marchal, Yves Robert

April 2004

Abstract

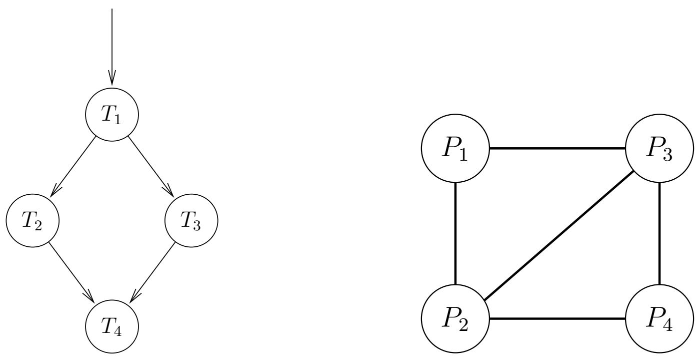
In this paper, we consider steady-state scheduling techniques for mapping a collection of application graphs onto heterogeneous systems, such as clusters and grids. We advocate the use of steady-state scheduling to solve this difficult problem. While the most difficult instances are shown to be NP-complete, most situations of practical interest are amenable to a periodic solution which can be described in compact form (polynomial size) and is asymptotically optimal.

Keywords: Mixed parallelism, Scheduling, steady-state, heterogeneous platforms

Résumé

Dans ce rapport, nous appliquons des techniques d'ordonnancement en régime permanent pour ordonnancer une suite de graphes d'application sur une plate-forme hétérogène de type *"grille de calcul"*. Nous prônons l'usage de l'ordonnancement en régime permanent pour résoudre ce problème difficile. Nous montrons que les instances les plus ardues de ce problème sont NP-complètes, alors que la plupart des situations présentant un intérêt pratique peuvent être résolues par une solution périodique qui admet une description compacte (de taille polynomiale) et qui est asymptotiquement optimale.

Mots-clés: Parallélisme mixte, Ordonnancement, régime permanent, plates-formes hétérogènes



(a) Application graph (b) Platform graph

Figure 1: The application and platform graphs

1 Introduction

The traditional objective of scheduling algorithms is makespan minimization: given a task graph and a set of computing resources, find a mapping of the tasks onto the processors, and order the execution of the tasks so that: (i) task precedence constraints are satisfied; (ii) resource constraints are satisfied; and (iii) a minimum schedule length is provided. However, makespan minimization turned out to be NP-hard in most practical situations [37, 1]. The advent of more heterogeneous architectural platforms is likely to even increase the computational complexity of the process of mapping applications to machines.

An idea to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective. Instead of aiming at the absolute minimization of the execution time, why not consider asymptotic optimality? After all, the number of tasks to be executed on the computing platform is expected to be very large: otherwise why deploy the corresponding application on computational grids? To state this informally: if there is a nice (meaning, polynomial) way to derive, say, a schedule whose length is two hours and three minutes, as opposed to an optimal schedule that would run for only two hours, we would be satisfied.

This approach has been pioneered by Bertsimas and Gamarnik [9]. Steady-state scheduling allows to relax the scheduling problem in many ways. Initialization and clean-up phases are neglected. The initial integer formulation is replaced by a continuous or rational formulation. The precise ordering and allocation of tasks and messages are not required, at least in the first step. The main idea is to characterize the activity of each resource during each time-unit: which (rational) fraction of time is spent computing, which is spent receiving or sending to which neighbor. Such activity variables are gathered into a linear program, which includes conservation laws that characterize the global behavior of the system.

In this paper, we consider the execution of a complex application on heterogeneous computing platforms. The complex application consists of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks, modeled as an *application graph*. There are dependences (precedence constraints) between these tasks. A typical example is the repeated execution of the same algorithm on several distinct data samples. Consider the simple application graph depicted in Figure 1(a). This graph models the algorithm. There is a main loop which is executed several times. Within each loop iteration, there are four tasks to be performed. Each loop iteration is what we call a problem instance. Each problem instance operates on different data, but all instances share the same *application graph*, i.e. the acyclic graph of Figure 1(a). For each node (task type) in the application graph, there are N distinct tasks to be executed, where N is the number of iterations in the main loop. Similarly, for each edge (file type) in the application graph, there are N different files to be transmitted.

We use another graph, the *platform graph*, for the grid platform. We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of an undirected graph. See the example in Figure 1(b) with four processors and five communication links. Each node is a computing resource (a processor, or a cluster, or even a router with no computing capabilities) capable of computing and/or communicating with its neighbors at (possibly) different rates. The underlying interconnection network may be very complex and, in particular, may include multiple paths and cycles (just as the Ethernet does).

Because the problem instances to be solved are independent, their execution can be pipelined. At a given time-step, different processors may well compute different tasks belonging to different problem instances. In the example, a given processor P_i may well compute the tenth copy of task T_1, corresponding to problem number 10, while another processor P_j computes the eighth copy of task T_3, which corresponds to problem number 8. However, because of the dependence constraints, note that P_j could not begin the execution of the tenth copy of task T_3 before that P_i has terminated the execution of the tenth copy of task T_1 and sent the required data to P_j (if $i \neq j$).

Deriving a steady-state solution for this complex mapping problem amounts to characterize the usage of processors and communication links: for a given processor, which fraction of time is spent executing which task type? for a given communication link, which fraction of time is spent communicating which file type? The objective is to maximize the throughput, which is the number of problem instances solved per time-unit, i.e. the number of copies of the application graph which are processed per time-unit. Of course, the previous activity fractions and the throughput are rational numbers. To derive the actual periodic schedule there will remain to scale everything, so as to derive an integer time-period. But the beauty of steady-state scheduling is that this reconstruction can be automatically computed from the rational values. We can derive a periodic schedule and express it in compact form, contrarily to the traditional makespan minimization approach which would require a scheduling date for all tasks and files.

The objective of the paper is to assess the limits of steady-state scheduling when applied to the difficult mapping problem that we just described. When is this approach asymptotically optimal? What is the inherent complexity of computing the optimal steady-state throughput? The first major contribution of the paper is a complexity result assessing that the most general instance of the problem is NP-complete. Showing that the problem does belong to the class NP (i.e. that a solution can be verified in polynomial time) already is a challenging problem. The second major result is positive: the optimal steady-state throughput can be computed in polynomial time for most practical instances, and the corresponding actual schedule is asymptotically optimal.

The rest of the paper is organized as follows. In Section 2, we introduce our base model of computation and communication, and we formally state the steady-state scheduling problem to be solved. Then we deal with problems of increasing difficulty:

- We start (Section 3) with the most simplified version of the problem, in order to introduce the main ideas: we address the case where the application graph is reduced to a single node. In Section 3.1, we use a linear programming approach to derive the optimal throughput. We give an algorithm to find a schedule that achieves this optimal throughput in Section 3.2.

- Then, we deal with arbitrary application graphs in Section 4. In Section 4.2, we provide the optimal solution to this problem, using the same kind of method as in Section 3.1. We give an algorithm to find a schedule that achieves this optimal throughput in Section 4.3. Section 4.5 states the complexity of the previous method and gives some insights for a practical implementation.

- In Section 5, we prove that the most general instance of the problem is NP-Complete.

- At last, in Section 6, we describe some related works and we give some remarks and conclusions in Section 7.

2 Models

2.1 The application

The application is a suite of problem instances, each instance being modeled by the same *application graph*. More precisely:

- Let $\mathcal{P}^{(1)}, \mathcal{P}^{(2)}, \ldots, \mathcal{P}^{(N)}$ be the N problems to solve, where N is large.

- Each problem $\mathcal{P}^{(m)}$ corresponds to a copy $G_A^{(m)} = (V_A^{(m)}, E_A^{(m)})$ of the same *application graph* $G_A = (V_A, E_A)$. The number $|V_A|$ of nodes in G_A is the number of task types. In the example of Figure 1(a), there are four task types, denoted as T_1, T_2, T_3 and T_4.

- Overall, there are $N.|V_A|$ tasks to process, since there are N copies of each task type.

For technical reasons it is simpler to have a single input task (a task without any predecessor) and a single output task (a task without any successor) in the application graph. To this purpose, we introduce two fictitious tasks, T_{begin} which is connected to the roots of the application graph and accounts for distributing the input files, and T_{end} which is connected to every task with no successor in the graph and accounts for gathering the output files.

2.2 The architecture

The target heterogeneous platform is represented by a directed graph, the *platform graph* $G_P = (V_P, E_P)$. More precisely:

- There are $p = |V_P|$ nodes $P_1, P_2, \dots, P_p$ in V_P that represent the processors. In the example of Figure 1(b) there are four processors, hence $p = 4$. See below for processor speeds and execution times.

- Each edge represents a physical interconnection. Each edge $e_{ij} \in E_P : P_i \to P_j$ is labeled by a value $c_{i,j}$ which represents the time to transfer a message of unit length between P_i and P_j, in either direction: we assume that the link between P_i and P_j is bidirectional and symmetric. A variant would be to assume two unidirectional links, one in each direction, with possibly different label values. If there is no communication link between P_i and P_j we let $c_{i,j} = +\infty$, so that $c_{i,j} < +\infty$ means that P_i and P_j are neighbors in the communication graph. With this convention, we can assume that the interconnection graph is (virtually) complete.

- We assume a *full overlap, single-port* operation mode, where a processor node can simultaneously receive data from one of its neighbor, perform some (independent) computation, and send data to one of its neighbor. At any given time-step, there are at most two communications involving a given processor, one in emission and the other in reception. Other models have been considered in [6, 4].

2.3 Execution times

- Processor P_i requires $w_{i,k}$ time units to process a task of type T_k.

- Note that this framework is quite general, because each processor has a different speed for each task type, and these speeds are not related: they are *inconsistent* with the terminology of [14]. Of course, we can always simplify the model. For instance we can assume that $w_{i,k} = w_i \times \delta_k$, where w_i is the inverse of the relative speed of processor P_i, and δ_k the weight of task T_k. Finally, note that routers can be modeled as nodes with no processing capabilities.

Because the task T_{begin} is fictitious, we let $w_{i,begin} = 0$ for each processor P_i holding the input files and $w_{i,begin} = +\infty$ otherwise.

Using T_{end}, we model two different situations: either the results (the output files of the tree leaves) do not need to be gathered and should stay in place, or all the output files have to be gathered to a particular processor P_{dest} (for visualization or post processing, for example).

In the first situation (output files should stay in place), no file of type $e_{k,end}$ is sent between any processor pair, for each edge $e_{k,end} : T_k \to T_{end}$. It is modeled by letting $data_{k,end} = +\infty$ (see below the definition of $data_{k,l}$).

In the second situation, where results have to be collected on a single processor P_{dest} then, we let $w_{dest,end} = 0$ (on the processor that gathers the results) and $w_{i,end} = +\infty$ on the other processors. Files of type $e_{k,end}$ can be sent between any processor pair since they have to be transported to P_{dest}.

2.4 Communication times

- Each edge $e_{k,l} : T_k \rightarrow T_l$ in the application graph is weighted by a communication cost $data_{k,l}$ that depends on the tasks T_k and T_l. It corresponds to the amount of data output by T_k and required as input to T_l.
- Recall that the time needed to transfer a unit amount of data from processor P_i to processor P_j is $c_{i,j}$. Thus, if a task $T_k^{(m)}$ is processed on P_i and task $T_l^{(m)}$ is processed on P_j, the time to transfer the data from P_i to P_j is equal to $data_{k,l} \times c_{i,j}$; this holds for any edge $e_{k,l} : T_k \rightarrow T_l$ in the application graph and for any processor pair P_i and P_j. Again, once a communication from P_i to P_j is initiated, P_i (resp. P_j) cannot handle a new emission (resp. reception) during the next $data_{k,l} \times c_{i,j}$ time units.

2.5 Allocations and cyclic schedules

We need the following definitions: allocation, schedule, makespan, cyclic schedule, K-periodic schedule, throughput. Although some are fairly intuitive, we formally state them all. The important result is Theorem 1, but the proof is technical (and boring).

Definition 1 (Allocation). *A valid allocation is a pair of mappings $\pi : V_A \mapsto V_P$ and $\sigma : E_A \mapsto \{\text{paths in } G_P\}$ such that for each edge $e_{k,l} : T_k \rightarrow T_l$:*

$$\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \dots, P_{i_p}) \text{ with } \begin{cases} P_{i_1} = \pi(T_k), P_{i_p} = \pi(T_l) \text{ and} \\ (P_{i_j} \rightarrow P_{i_{j+1}}) \in E_P \text{ for all } j \in [\![1, p-1]\!] \end{cases} .$$

Obviously, π maps the tasks of the application graph G_A onto the platform nodes and σ maps the (files associated to the) edges of G_A onto the paths of the platform network.

Definition 2 (Schedule). *A valid schedule associated to a valid allocation (π, σ) is a pair of mappings $t_\pi : V_A \mapsto \mathbb{Q}$ and $t_\sigma : E_A \times E_P \mapsto \mathbb{Q}$ satisfying to the following constraints:*

- ***precedence:*** *For all $e_{k,l} : T_k \rightarrow T_l$, if $\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \dots, P_{i_p})$ then*

$$\begin{aligned} t_\pi(T_k) + w_{i_1,k} &\leqslant t_\sigma(e_{k,l}, P_{i_1} \rightarrow P_{i_2}) \\ t_\sigma(e_{k,l}, P_{i_1} \rightarrow P_{i_2}) + data_{k,l} \times c_{i_1,i_2} &\leqslant t_\sigma(e_{k,l}, P_{i_2} \rightarrow P_{i_3}) \\ &\vdots \\ t_\sigma(e_{k,l}, P_{i_{p-1}} \rightarrow P_{i_p}) + data_{k,l} \times c_{i_{p-1},i_p} &\leqslant t_\pi(T_l) \end{aligned}$$

- ***computations:*** *A processor cannot execute two tasks simultaneously: for all $T_k \neq T_l$, we have*

$$\pi(T_k) = \pi(T_l) \Rightarrow [t_\pi(T_k), t_\pi(T_k) + w_{\pi(T_k),k}[\cap [t_\pi(T_l), t_\pi(T_l) + w_{\pi(T_l),l}[= \emptyset$$

- ***communications:*** *Two different files cannot simultaneously circulate on the edge from P_i to P_j: for all $e_{k_1,l_1} \neq e_{k_2,l_2} \in E_A$ and for all $P_i \rightarrow P_j$ we have*

$$\begin{aligned} &[t_\sigma(e_{k_1,l_1}, P_i \rightarrow P_j), t_\sigma(e_{k_1,l_1}, P_i \rightarrow P_j) + data_{k_1,l_1} \times c_{i,j}[\\ &\qquad \cap [t_\sigma(e_{k_2,l_2}, P_i \rightarrow P_j), t_\sigma(e_{k_2,l_2}, P_i \rightarrow P_j) + data_{k_2,l_2} \times c_{i,j}[= \emptyset \end{aligned}$$

- ***one-port for outputs*** *A processor cannot send messages to two processors simultaneously: for all $e_{k_1,l_1}, e_{k_2,l_2} \in E_A$ and for all $(P_i \rightarrow P_{j_1}) \neq (P_i \rightarrow P_{j_2})$ we have*

$$\begin{aligned} &[t_\sigma(e_{k_1,l_1}, P_i \rightarrow P_{j_1}), t_\sigma(e_{k_1,l_1}, P_i \rightarrow P_{j_1}) + data_{k_1,l_1} \times c_{i,j_1}[\\ &\qquad \cap [t_\sigma(e_{k_2,l_2}, P_i \rightarrow P_{j_2}), t_\sigma(e_{k_2,l_2}, P_i \rightarrow P_{j_2}) + data_{k_2,l_2} \times c_{i,j_2}[= \emptyset \end{aligned}$$

- ***one-port for inputs*** *A processor cannot receive messages from two processors simultaneously: for all* $e_{k_1,l_1}, e_{k_2,l_2} \in E_A$ *and for all* $(P_{i_1} \to P_j) \neq (P_{i_2} \to P_j)$ *we have*

$$[t_\sigma(e_{k_1,l_1}, P_{i_1} \to P_j), t_\sigma(e_{k_1,l_1}, P_{i_1} \to P_j) + data_{k_1,l_1} \times c_{i_1,j}[\\ \cap\, [t_\sigma(e_{k_2,l_2}, P_{i_2} \to P_j), t_\sigma(e_{k_2,l_2}, P_{i_2} \to P_j) + data_{k_2,l_2} \times c_{i_2,j}[= \emptyset$$

Definition 3 (Makespan of a schedule). *The makespan of a valid schedule* (t_π, t_σ) *is*

$$\max_{T_k \in V_A} \left(t_\pi(T_k) + w_{\pi(T_k),k}\right) - \min_{T_k \in V_A} t_\pi(T_k)$$

The next definitions are more technical:

Definition 4 (Extended graph). *Let* $G_A = (V_A, E_A)$ *be an application graph and* S *a set. We write:*

$$\begin{aligned} V_A \otimes S &= V_A \times S \\ E_A \otimes S &= \left\{\left((T_k, n), (T_l, n)\right) \middle| e_{k,l} : T_k \to T_l \in E_A \text{ et } n \in S\right\} \\ G_A \otimes S &= (V_A \otimes S, E_A \otimes S) \end{aligned}$$

$G_A \otimes S$ *is the extended graph of* G_A *along* S*.*

Definition 5 (Cyclic schedule). *A cyclic schedule of an application graph* G_A *on a platform graph* G_P *is a valid scheduling of* $G_A \otimes \mathbb{N}$ *on* G_P*.*

Definition 6 (Makespan of a cyclic schedule). *The makespan of the first* N *tasks of a cyclic schedule is*

$$D_N = \max_{(T_k,n) \in V_A \times [\![1,N]\!]} \left(t_\pi(T_k, n) + w_{\pi(T_k,n),k}\right) - \min_{(T_k,n) \in V_A \times [\![1,N]\!]} t_\pi(T_k, n)$$

Definition 7 (*K*-periodic sequence). *A sequence* u *is* K*-periodic if it is nondecreasing and if there exists an integer* n_0 *and a rational* $T_p > 0$ *such that*

$$\forall n \geqslant n_0 : u_{n+K} = u_n + T_p$$

K *is the periodicity factor,* T_p *is the period, and* $\frac{K}{T_p}$ *is the throughput of the sequence. If* $n_0 = 0$*, we say that the sequence is* strictly K*-periodic.*

Definition 8 (*K*-periodic schedule). *To simplify the notations,* $f(x, n)$ *and* $f(x)(n)$ *are considered as equivalent. A cyclic schedule* (t_π, t_σ) *is* K*-periodic with period* T_p *if for all* $T_k \in V_A$*, the sequence* $t_\pi(T_k)$ *is strictly* K*-periodic with period* T_p*, and if for all* $e_{k,l} \in E_A$ *and for all* $P_i \to P_j$*,* $t_\sigma(e_{k,l}, P_i \to P_j)$ *is strictly* K*-periodic with period* T_p*.*

Intuitively, this amounts to say that the scheduling of K consecutive instances of the application graph is periodic.

Definition 9 (Throughput of a cyclic schedule). *The throughput of a cyclic schedule is defined as the following limit, if it exists:*

$$\lim_{N \to \infty} \frac{N}{D_N}$$

Remark 1. *A* K*-periodic schedule of period* T_p *is fully characterized by the first* K *values of the* t_π (T_k) *and of the* t_σ $(e_{k,l}, P_i \to P_j)$*, and its throughput is* $\frac{K}{T_p}$*.*

The previous remark leads to an important question: is there a one-to-one correspondence between K periodic schedules of period T_p and "patterns" of length T_p for the execution of K consecutive instances of the application graph? The answer is positive, and this is good news: if we show that resource constraints are satisfied during the period, then this guarantees the existence of a valid schedule with the desired throughput. This result is formally proved in the rest of the section, which can be omitted by the reader.

2.6 Patterns and periodic schedules

Definition 10 (K-pattern of length T_p). *A K-pattern of length T_p is defined by an allocation (π, σ) from $G_A \otimes [\![1, K]\!]$ onto G_P, and by two mappings $\tilde{t}_\pi : V_A \otimes [\![1, K]\!] \mapsto [0, T_p[$ and $\tilde{t}_\sigma : (E_A \otimes [\![1, K]\!]) \times E_P \mapsto [0, T_p[$ satisfying all resource constraints and all one-port constraints modulo T_p.*

Definition 11 (M_{K,T_p}). *M_{K,T_p} denotes the canonical mapping from the set of K-periodic schedules of period T_p onto the set of K-pattern of length T_p: we let $M_{K,T_p}(t_\pi, t_\sigma) = (\tilde{t}_\pi, \tilde{t}_\sigma)$ such that:*

$$\forall T_k \in V_A, \forall n \in [\![1, K]\!] : \tilde{t}_\pi(T_k, n) = t_\pi(T_k, n) \mod T_p$$
$$\forall e_{k,l} \in E_A, \forall P_i \to P_j \in E_P, \forall n \in [\![1, K]\!] : \tilde{t}_\sigma(e_{k,l}, n, P_i \to P_j) = t_\sigma(e_{k,l}, n, P_i \to P_j) \mod T_p$$

Lemma 1 (M_{1,T_p} is surjective). *Let $T_p \in \mathbb{Q}_+^*$. Let (π, σ) be an allocation from G_A onto G_P. Given $\tilde{t}_\pi : V_A \mapsto [0, T_p[$ and $\tilde{t}_\sigma : E_A \times E_P \mapsto [0, T_p[$ satisfying all resource constraints and all one-port constraints modulo T_p, there exists a 1-periodic schedule (t_π, t_σ) from G_A onto G_P (hence also satisfying to all precedence constraints), of period T_p, such that*

$$\forall T_k \in V_A, \forall n \in \mathbb{N} : t_\pi(T_k, n) = \tilde{t}_\pi(T_k) \mod T_p$$
$$\forall e_{k,l} \in E_A, \forall P_i \to P_j \in E_P, \forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma(e_{k,l}, P_i \to P_j) \mod T_p$$

Proof. We look for t_π and t_σ expressed as

$$t_\pi(T_k, n) = \tilde{t}_\pi(T_k) + (n + \Delta_\pi(T_k))T_p$$
$$t_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma(e_{k,l}, P_i \to P_j) + (n + \Delta_\sigma(e_{k,l}, P_i \to P_j))T_p,$$

where Δ_σ and Δ_π take their values in $\mathbb{Z}$. We give necessary and sufficient conditions on Δ_σ and Δ_π for (t_π, t_σ) to be a 1-periodic schedule from G_A onto G_P.

- **Precedence constraints:** Let $e_{k,l} \in E_A$, and $\sigma(e_{k,l}) = (P_{i_1}, P_{i_2}, \ldots, P_{i_p})$.

$$\forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_{i_{p-1}} \to P_{i_p}) + data_{k,l} \times c_{i_{p-1},i_p} \leqslant t_\pi(T_l, n)$$
$$\Leftrightarrow$$
$$\tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) + \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p})T_p + data_{k,l} \times c_{i_{p-1},i_p} \leqslant \tilde{t}_\pi(T_l) + \Delta_\pi(T_l)T_p$$
$$\Leftrightarrow$$
$$\tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) + \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p})T_p + data_{k,l} \times c_{i_{p-1},i_p} \leqslant \tilde{t}_\pi(T_l) + \Delta_\pi(T_l)T_p$$
$$\Leftrightarrow$$
$$\Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) \leqslant \left\lfloor \frac{\tilde{t}_\pi(T_l) - \tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - data_{k,l} \times c_{i_{p-1},i_p}}{T_p} \right\rfloor + \Delta_\pi(T_l)$$

  Similarly, we have

$$\forall n \in \mathbb{N} : t_\sigma((e_{k,l}, n), P_{i_1} \to P_{i_2}) + data_{k,l} \times c_{i_1,i_2} \leqslant t_\sigma((e_{k,l}, n), P_{i_2} \to P_{i_3})$$
$$\Leftrightarrow$$
$$\Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) \leqslant \left\lfloor \frac{\tilde{t}_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) - \tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - data_{k,l} \times c_{i_1,i_2}}{T_p} \right\rfloor + \Delta_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}),$$

  and

$$\forall n \in \mathbb{N} : t_\pi(T_k) + w_{i_1,k} \leqslant t_\sigma(e_{k,l}, P_{i_1} \to P_{i_2})$$
$$\Leftrightarrow$$
$$\Delta_\pi(T_k) \leqslant \left\lfloor \frac{\tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \tilde{t}_\pi(T_k) - w_{i_1,k}}{T_p} \right\rfloor + \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2})$$

(t_π, t_σ) satisfies all precedence constraints if and only if the following system of potential constraints has a solution:

$$\forall e_{k,l} : T_k \to T_l$$
$$\begin{cases} \Delta_\pi(T_k) - \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) \leqslant \left\lfloor \dfrac{\tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \tilde{t}_\pi(T_k) - w_{i_1,k}}{T_p} \right\rfloor \\ \Delta_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - \Delta_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) \leqslant \left\lfloor \dfrac{\tilde{t}_\sigma(e_{k,l}, P_{i_2} \to P_{i_3}) - \tilde{t}_\sigma(e_{k,l}, P_{i_1} \to P_{i_2}) - data_{k,l} \times c_{i_1,i_2}}{T_p} \right\rfloor \\ \vdots \\ \Delta_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - \Delta_\pi(T_l) \leqslant \left\lfloor \dfrac{\tilde{t}_\pi(T_l) - \tilde{t}_\sigma(e_{k,l}, P_{i_{p-1}} \to P_{i_p}) - data_{k,l} \times c_{i_{p-1},i_p}}{T_p} \right\rfloor \end{cases}$$

The graph G_A is acyclic, hence the potential graph associated to the previous equations is acyclic too. In particular, there are no cycles of negative weight, and the system admits a solution that can be computed in polynomial time [16].

- **Resource constraints:** For all $I_1, I_2 \subset [0, T_p[$, we have $(I_1 + T_p\mathbb{Z}) \cap (I_2 + T_p\mathbb{Z}) = (I_1 \cap I_2) + T_p\mathbb{Z}$. Since $(\tilde{t}_\pi,\tilde{t}_\sigma)$ satisfies to all resource constraints, the same holds true for (t_π, t_σ).
- **1-periodicity of period T_p:** for all T_k, $t_\pi(T_k)$ is 1-periodic of period T_p and for all $e_{k,l}$, $t_\sigma(e_{k,l})$ also is 1-periodic of period T_p. ■

Lemma 2. *If M_{K_1,T_p} et M_{K_2,T_p} are surjective, then $M_{K_1+K_2,T_p}$ is surjective too.*

Proof. Let $(\tilde{t}_\pi, \tilde{t}_\sigma)$ be $(K_1 + K_2)$-pattern of length T_p. Then we define $\tilde{t}^{(1)}_\pi : V_A \times [\![1, K_1]\!] \mapsto [0, T_p[$ as $\tilde{t}^{(1)}_\pi(T_k, n) = \tilde{t}_\pi(T_k, n)$ and $\tilde{t}^{(1)}_\sigma : (E_A \times [\![1, K]\!]) \times E_P \mapsto [0, T_p[$ as $\tilde{t}^{(1)}_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma((e_{k,l}, n), P_i \to P_j)$; then $(\tilde{t}^{(1)}_\pi, \tilde{t}^{(1)}_\sigma)$ is a K_1-pattern of length T_p. Similarly, we define $\tilde{t}^{(2)}_\pi : V_A \times [\![1, K_2]\!] \mapsto [0, T_p[$ as $\tilde{t}^{(2)}_\pi(T_k, n) = \tilde{t}_\pi(T_k, K_1 + n)$ and $\tilde{t}^{(2)}_\sigma : (E_A \times [\![1, K_2]\!]) \times E_P \mapsto [0, T_p[$ as $\tilde{t}^{(2)}_\sigma((e_{k,l}, n), P_i \to P_j) = \tilde{t}_\sigma((e_{k,l}, K_2 + n), P_i \to P_j)$; then $(\tilde{t}^{(2)}_\pi, \tilde{t}^{(2)}_\sigma)$ is a K_2-pattern of length T_p.

Since M_{K_1,T_p} et M_{K_2,T_p} are surjective, let $(t^{(1)}_\pi, t^{(1)}_\sigma)$ (resp. $(t^{(2)}_\pi, t^{(2)}_\sigma)$) be a schedule of pattern $(\tilde{t}^{(1)}_\pi, \tilde{t}^{(1)}_\sigma)$ (resp. $(\tilde{t}^{(2)}_\pi, \tilde{t}^{(2)}_\sigma)$). Then define $t_\pi : V_A \times \mathbb{N} \mapsto \mathbb{Q}$ as

$$t_\pi(T_k, n) = \begin{cases} t^{(1)}_\pi(T_k, n) & \text{if } n \in [\![1, K_1]\!] + (K_1 + K_2)\mathbb{Z} \\ t^{(2)}_\pi(T_k, n) & \text{otherwise} \end{cases}$$

and define $t_\sigma : (E_A \times \mathbb{N}) \times E_P \mapsto \mathbb{Q}$ as

$$t_\sigma((e_{k,l}, n), P_i \to P_j) = \begin{cases} t^{(1)}_\sigma((e_{k,l}, n), P_i \to P_j) & \text{if } n \in [\![1, K_1]\!] + (K_1 + K_2)\mathbb{Z} \\ t^{(2)}_\sigma((e_{k,l}, n), P_i \to P_j) & \text{otherwise} \end{cases},$$

We have derived a $(K_1 + K_2)$-cyclic schedule of period T_p and of pattern $(\tilde{t}_\pi, \tilde{t}_\sigma)$. ■

Theorem 1. *For all $K \in \mathbb{N}^*$, M_{K,T_p} is surjective.*

Proof. By straightforward induction, using the previous two lemmas. ■

2.7 From allocations to periodic schedule

We are ready for the main result of this section, which captures a very important feature of steady-state scheduling, namely the guarantee that it is always possible to reconstruct a periodic schedule from a set of weighted allocations satisfying all resource (computation and communication) constraints.

Intuitively, we aim at "mapping" instances of the application graph onto the platform graph. An allocation simply is the recording of one possible mapping (i.e. where everything takes place, from computations to file transfers). Interleaving several different allocations looks promising to squeeze the most out of the platform graph. Distinct allocations may use different resources, and their simultaneous use is easy in that case; but they may also share some resources, and we then have to ensure that those shared resources are accessed in exclusive mode. In other words, we have to orchestrate resource utilization requested by different allocations to reconstruct a valid periodic schedule. Given a set of allocations, each of them having to be used with a prescribed frequency, is it possible to reconstruct a valid periodic schedule with optimal throughput? The answer is positive, and we state it now formally. Consider an application graph $G_A = (V_A, E_A)$ and a platform graph $G_P = (V_P, E_P)$:

Definition 12 (Weight of an allocation). *The weight of an allocation $\mathcal{A} = (\pi, \sigma)$ is a positive rational number α representing the number of times that the allocation is to be used every time-unit.*

Definition 13 (Resource-compliant weighted allocations). *A set of r weighted allocations $(\mathcal{A}_m, \alpha_m)_{1\leqslant m\leqslant r}$, where $\mathcal{A}_m = (\pi_m, \sigma_m)$, is resource-compliant if $\forall P_i \in V_P$ the following three relations hold:*

- ***Computations:*** $\displaystyle\sum_{m=1}^{r}\left(\sum_{T_k/\pi_m(T_k)=i} w_{i,k}.\alpha_m\right) \leqslant 1$
- ***Incoming communications:*** $\displaystyle\sum_{m=1}^{r}\left(\sum_{e_{k,l}\in E_A}\left(\sum_{(P_j\to P_i)\in\sigma_m(e_{k,l})} data_{k,l}.c_{j,i}.\alpha_m\right)\right) \leqslant 1$
- ***Outgoing communications:*** $\displaystyle\sum_{m=1}^{r}\left(\sum_{e_{k,l}\in E_A}\left(\sum_{(P_i\to P_j)\in\sigma_m(e_{k,l})} data_{k,l}.c_{i,j}.\alpha_m\right)\right) \leqslant 1$

Basically, the first constraint states that within a time-unit, each processor can compute as many tasks as required by each allocation counted as many times as its weight. The second and third constraint are the counterpart for incoming and outgoing communications under the one-port model.

Theorem 2. *Given a set of r weighted allocations $(\mathcal{A}_m, \alpha_m)$, we can reconstruct a valid periodic schedule of throughput $\rho = \sum_{m=1}^{r} \alpha_m$.*

The significance of Theorem 2 is that it allows to go from purely local constraints to a global steady-state schedule. We provide a formal proof below but the reader may want to skip it and follow the derivation conducted in Section 3.2 for a simple particular case (where the application graph is reduced to one input task, one main task and one output task). Section 3.2 can be read independently of the proof below.

Proof. For each weighted allocation $(\mathcal{A}_m, \alpha_m)$, let $s(P_i \to P_j, e_{k,l}, \mathcal{A}_m)$ denote the fraction of time spent by P_i to send to P_j data involved by the edge $e_{k,l}$ for the allocation. We have $s(P_i \to P_j, e_{k,l}, \mathcal{A}_m) = data_{k,l}.c_{i,j}.\alpha_m$ if the edge $P_i \to P_j$ is used to transmit files of type $e_{k,l}$ by the allocation $\mathcal{A}_m$ (that is if $P_i \to P_j \in \sigma_m(e_{k,l})$), and $s(P_i \to P_j, e_{k,l}, \mathcal{A}_m) = 0$ otherwise.

Bipartite graph of the communications We build a weighted bipartite multigraph $G_B = (V_B, E_B)$ as follows: for each node P_i in G_P, we create two nodes P_i^{send} and P_i^{recv}, so that $|V_B| = 2|V_P|$. For each non-zero value of $\sum_{k,l,m} s(P_i \to P_j, e_{k,l}, \mathcal{A}_m)$ we insert an edge between P_i^{send} and P_j^{recv}, whose weight is $\sum_{k,l,m} s(P_i \to P_j, e_{k,l}, \mathcal{A}_m)$, i.e. the overall communication time between P_i and P_j. The number of edges in G_B is such that $|E_B| \leqslant |E_P|$. We are looking for a decomposition of the bipartite multigraph G_B into a set of subgraphs where a node (sender or receiver) is occupied by at most one communication. This means that at most one edge reaches each node in the subgraph. In other words, only communications corresponding to a matching in the bipartite graph can be performed simultaneously, and the desired decomposition of the graph is in fact an edge coloring.

The total weight of a node P_i^{send}, defined as the sum of the weight of its incident edges, does not exceed 1 because of the constraint of the incoming communications. Similarly, the total weight of a node P_i^{recv} is not larger than 1.

Decomposition into matchings The weighted edge coloring algorithm for bipartite graphs of [34, vol.A chapter 20] provides a polynomial number of weighted matchings $(M_1, \gamma_1), \ldots, (M_s, \gamma_s)$ such that $\chi_{G_B} = \sum_u \gamma_u . \chi_u$ and $\sum_u \gamma_u \leqslant 1$. Here, χ_u denotes the characteristic function of the matching ($\chi_u(e) = 1$ iff edge e belongs to the matching M_u). The number of matchings is bounded by $|E_B|$ and the complexity of the algorithm is $O(|E_B|^2)$, hence polynomial in the size of the application and platform graphs, and of the number of allocations.

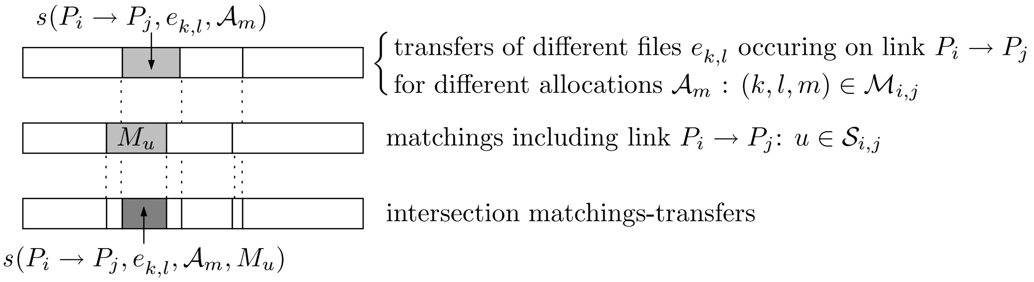
Link between matchings and allocations For each edge $P_i \rightarrow P_j$, the sum of the weight of the matching containing this edge is equal to the weight of this edge in the bipartite graph. Let us call $\mathcal{M}_{i,j}$ the set of matchings including edge $P_i \rightarrow P_j$ and $\mathcal{S}_{i,j}$ the set of values (k, l, m) such that $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m) \neq 0$. We have:

$$\sum_{u \in \mathcal{M}_{i,j}} \gamma_u = \sum_{(k,l,m) \in \mathcal{S}_{i,j}} s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m)$$

which is also:

$$\sum_{u \in \mathcal{M}_{i,j}} \gamma_u = \sum_{(k,l,m) \in \mathcal{S}_{i,j}} data_{k,l} \times c_{i,j} \times \alpha_m$$

The time intervals of length γ_u involved in the left sum are represented on the first line of the following figure, whereas the intervals of length $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m)$ appearing in the second sum are represented on the second line. We have to split both types of intervals into sub-intervals so that each of them is at the same time part of a single interval of the first sum and part of a single interval of the second sum.



The above figure presents a simple way to split and distribute the communications of the intervals $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m)$ into the matchings. We denote by $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u)$ the fraction of $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m)$ corresponding to the matching M_u in the distribution. As $data_{k,l}$ and $c_{i,j}$ are integers, if we let $\gamma_u = \frac{a_u}{b_u}$, $\forall m$, $\alpha_m = \frac{c_m}{d_m}$ where a_u, b_u, c_m, d_m are integers, the size of all chunks obtained in the decomposition of the previous figure is a multiple of $1/P$ where P is the least common denominator of all b_u's and d_u's. In other words, $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u)$ is a multiple of $\frac{1}{\mathrm{lcm}_u\{b_u\} \times \mathrm{lcm}_m\{d_m\}}$.

Building a periodic schedule Let us define $B = \mathrm{lcm}_u\{b_u\} \times \mathrm{lcm}_m\{d_m\} \times \mathrm{lcm}_{k,l}\{data_{k,l}\} \times \mathrm{lcm}_{i,j}\{c_{i,j}\}$. In what follows, we aim at building a pattern of length B such that all communications and all computations corresponding to $(\sum \alpha_m)B = \rho B$ application graphs can be realized ($\alpha_1 B$ according to allocation $\mathcal{A}_1$,..., and $\alpha_m B$ according to allocation $\mathcal{A}_m$). This amounts to construct a $\rho.B$-pattern of length B. Owing to Theorem 1, we know that the existence of such a pattern induces the existence of periodic schedule (that can be built in polynomial time) with period B and achieving ρB application graphs every B time units, thus the optimal throughput.

The period of size B will be decomposed in $(s+1)$ intervals of respective size $(\gamma_1 B, \ldots, \gamma_s B, (1 - \sum \gamma_u)B$, corresponding to the matchings. For each processor pair (P_i, P_j), the communications between P_i^{send} and P_j^{recv} will take place in time intervals $u \leqslant s$ such that $(P_i^{send}, P_j^{recv}) \in M_u$, so that during one interval, communications will take place independently, since one processor can be involved in at most one send and one receive operation. During the u^{th} interval, processor P_i sends to P_j exactly $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u) \frac{B}{data_{k,l} \times c_{i,j}}$ files of type $e_{k,l}$ corresponding to allocation $\mathcal{A}_m$ (thanks to the definition of B, this is a integer number of files). This transfer lasts $s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u) \times B$ time-units. The time needed to send all files for all allocations during the interval corresponding to matching M_u is:

$$\delta_u = \sum_{\mathcal{A}_m} \sum_{e_{k,l}} s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u) \times B = \gamma_u \times B$$

which is the length of the u^{th} interval. The total number of files $e_{k,l}$ sent by P_i to P_j for allocation $\mathcal{A}_m$ during one whole period is:

$$\sum_{u \in \mathcal{M}_{i,j}} s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m, M_u) \times \frac{B}{data_{k,l} \times c_{i,j}} = s(P_i \rightarrow P_j, e_{k,l}, \mathcal{A}_m) \times \frac{B}{data_{k,l} \times c_{i,j}} = \alpha_m \times B$$

so every files $e_{k,l}$ for allocation $\mathcal{A}_m$ may be transfered during one period of length B.

Computation Since communication and computations can be overlapped, computations will take place independently during the whole time period B. Let us now prove that processor P_i is able to process all the tasks assigned to it. The number of tasks of type T_k assigned to P_i is given by

$$W_{i,k} = (\sum_{m} \sum_{\pi_m(T_k)=i} \alpha_m)B,$$

and by construction of B, $W_{i,k}$ is an integer since $\forall m, \alpha_m B$ is an integer.
By construction,

$$\sum_{m} \sum_{\pi_m(T_k)=i} w_{i,k}.\alpha_m \leqslant 1,$$

so that

$$\sum_{m} \sum_{\pi_m(T_k)=i} w_{i,k}.(\alpha_m B) \leqslant B.$$

Thus, during the time period of duration B, P_i can process $\alpha_m B$ tasks of kind T_k, $\forall (k,m)$ such that $\pi_m(T_k) = i$. Thus, P_i is able to process all tasks assigned to it within the request time-frame.

Thus, it is possible to organize all the communications and all the computations corresponding to ρB application graphs in a time period of duration B. Owing to Theorem 1, there exist a periodic schedule (with period B) that achieves ρB application graphs every B time units.

Finally, define the size of the mapping problem I as the size of the application and platform graphs. If the number r of resource-compliant allocations is polynomial in I (which means that there is a reasonably bounded number of allocations), and their weights are polynomial in I (which means that their weight can be expressed in reasonable compact way) then $\log B$ has a size polynomial in I too, so that the construction and the description of the schedule are fully polynomial in I. We will use this property in the proof of Theorem 6. ∎

3 Steady-state scheduling of independent tasks

In this section we focus on scheduling independent tasks. Such an application is modeled with the very simple application graph depicted on Figure 2(a), which includes only three task types on a linear path: an input task, the main task and an output task. However, we point out that all the results shown in this section also hold true when the application graph is a tree.

3.1 Optimal throughput

In this section we derive a bound on the optimal throughput that can be achieved in steady-state mode, using a linear programming approach. Later in Section 3.2, we show how to build a periodic schedule that achieves this throughput. This is the beauty of the steady-state approach: all the information needed to construct the actual schedule lies in the solution of the linear program.

3.1.1 Steady-state activity variables

- For each edge $e_{k,l} : T_k \to T_l$ in the task graph and for each processor pair (P_i, P_j), we denote by $s(P_i \to P_j, e_{k,l})$ the (average) fraction of time spent each time-unit by P_i to send to P_j data involved by the edge $e_{k,l}$. Of course $s(P_i \to P_j, e_{k,l})$ is a nonnegative rational number. Think of an edge $e_{k,l}$ as requiring a new file to be transferred from the output of each task $T_k^{(m)}$ processed on P_i to the input of each task $T_l^{(m)}$ processed on P_j. Let the (fractional) number of such files sent per time-unit be denoted as $sent(P_i \to P_j, e_{k,l})$. We have the relation:

$$s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j}) \tag{1}$$

  which states that the fraction of time spent transferring such files is equal to the number of files times the product of their size by the elemental transfer time of the communication link.

- For each task type $T_k \in V_A$ and for each processor P_i, we denote by $\alpha(P_i, T_k)$ the (average) fraction of time spent each time-unit by P_i to process tasks of type T_k, and by $cons(P_i, T_k)$ the (fractional) number of tasks of type T_k processed per time unit by processor P_i. We have the relation

$$\alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k} \tag{2}$$

3.1.2 Steady-state equations

We search for rational values of all the variables $s(P_i \to P_j, e_{k,l})$, $sent(P_i \to P_j, e_{k,l})$, $\alpha(P_i, T_k)$ and $cons(P_i, T_k)$. We formally state the first constraints to be fulfilled.

Activities during one time-unit All fractions of time spent by a processor to do something (either computing or communicating) must belong to the interval $[0, 1]$, as they correspond to the average activity during one time unit:

$$\forall P_i, \forall T_k \in V_A,\ 0 \leqslant \alpha(P_i, T_k) \leqslant 1 \tag{3}$$

$$\forall P_i, P_j, \forall e_{k,l} \in E_A,\ 0 \leqslant s(P_i \to P_j, e_{k,l}) \leqslant 1 \tag{4}$$

One-port model for outgoing communications Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i,\ \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1 \tag{5}$$

where $n(P_i)$ denotes the neighbors of P_i. Recall that we can assume a complete graph owing to our convention with the $c_{i,j}$.

One-port model for incoming communications Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i,\ \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1 \tag{6}$$

Note that $s(P_j \to P_i, e_{k,l})$ is indeed equal to the fraction of time spent by P_i to receive from P_j files of type $e_{k,l}$.

Full overlap Because of the full overlap hypothesis, there is no further constraint on $\alpha(P_i, T_k)$ except that

$$\forall P_i,\ \sum_{T_k \in V_A} \alpha(P_i, T_k) \leqslant 1 \tag{7}$$

3.1.3 Conservation laws

The last constraints deal with *conservation laws.* Consider a given processor P_i, and a given edge $e_{k,l}$ in the application graph. During each time unit, P_i receives from its neighbors a given number of files of type $e_{k,l}$: P_i receives exactly $\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l})$ such files. Processor P_i itself executes some tasks T_k, namely $cons(P_i, T_k)$ tasks T_k, thereby generating as many new files of type $e_{k,l}$.

What does happen to these files? Some are sent to the neighbors of P_i, and some are consumed by P_i to execute tasks of type T_l. We derive the equation:

$$\forall P_i, \forall e_{k,l} \in E_A : T_k \to T_l,$$
$$\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) = \sum_{P_j \in n(P_i)} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l) \quad (8)$$

It is important to understand that equation (8) really applies to the steady-state operation. At the beginning of the operation of the platform, only input tasks are available to be forwarded. Then some computations take place, and tasks of other types are generated. At the end of this initialization phase, we enter the steady-state: during each time-period in steady-state, each processor can simultaneously perform some computations, and send/receive some other tasks. This is why Equation (8) is sufficient, we do not have to detail which operation is performed at which time-step.

Finally, we point out that Equation (8) has been written in the most general setting (several masters, prescribed resources for output files, etc.). We have a simpler formulation if we assume a single master and no constraint on the localization of output files. In that case, let $sent(P_i \to P_j)$ be the number of files sent from P_i to P_j within a time-unit, and let $cons(P_i)$ be the number of tasks consumed by P_i within a time-unit. We derive the conservation law, which states that each task received is either consumed or forwarded to another resource:

$$\forall P_i, \sum_{P_j \to P_i} sent(P_j \to P_i) = cons(P_i) + \sum_{P_i \to P_j} sent(P_i \to P_j)$$

3.1.4 Bound on the optimal steady-state throughput

The equations listed in the previous section constitute a linear programming problem, whose objective function is the total throughput, i.e. the number of tasks T_{end} consumed within one time-unit:

$$\rho = \sum_{i=1}^{p} cons(P_i, T_{end}) \quad (9)$$

Indeed, each time a task T_{end} has been consumed, the dependence constraints imply that a whole instance of the application graph has been executed. Here is a summary of the linear program:

$$
\begin{array}{l}
\textsc{Maximize } \rho = \sum_{i=1}^{p} cons(P_i, T_{end}),\\
\textsc{under the consstraints}\\
\left\{
\begin{array}{ll}
(10a) & \forall P_i, \forall T_k \in V_A, 0 \leqslant \alpha(P_i, T_k) \leqslant 1\\
(10b) & \forall P_i \to P_j, \forall e_{k,l} \in E_A, 0 \leqslant s(P_i \to P_j, e_{k,l}) \leqslant 1\\
(10c) & \forall P_i \to P_j, \forall e_{k,l} \in E_A, s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \times (data_{k,l} \times c_{i,j})\\
(10d) & \forall P_i, \forall T_k \in V_A, \alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k}\\
(10e) & \forall P_i, \displaystyle\sum_{P_j \to P_i} \sum_{e_{k,l} \in E_A} s(P_j \to P_i, e_{k,l}) \leqslant 1\\
(10f) & \forall P_i, \displaystyle\sum_{P_i \to P_j} \sum_{e_{k,l} \in E_A} s(P_i \to P_j, e_{k,l}) \leqslant 1\\
(10g) & \forall P_i, \displaystyle\sum_{T_k \in V_A} \alpha(P_i, T_k) \leqslant 1\\
(10h) & \forall P_i, \forall e_{k,l} \in E_A : T_k \to T_l,\\
& \qquad \displaystyle\sum_{P_j \to P_i} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) =\\
& \qquad\qquad \displaystyle\sum_{P_i \to P_j} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)
\end{array}
\right.
\end{array}
\tag{10}
$$

As already pointed out, the beauty of steady-state scheduling is that the optimal throughput ρ given by the linear program can be achieved. We show how to build a periodic schedule of throughput ρ in Section 3.2, and we establish its asymptotic optimality among all possible schedules (not only periodic ones) in Section 3.3.

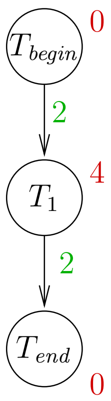
3.2 Reconstruction of an effective schedule

Consider the platform depicted on Figure 2(b) and a set of independent tasks (whose characteristics are depicted on Figure 2(a)). We suppose that all input files $e_{begin,1}$ initially reside on P_1, and that all output files $e_{1,end}$ have to be gathered on P_1. These conditions are ensured by imposing that neither T_{begin} nor T_{end} can be processed on another place than P_1.

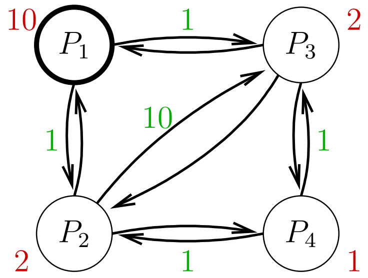
Solving the linear program, we get the values summarized on Figure 3. In Figure 3(b), solid edges represent the transfers of $e_{begin,1}$ and dashed ones the transfers of $e_{1,end}$. The weights of the edges represent the fraction of time spent for the associated file transfer. We have to show how to reconstruct a schedule from this description. More precisely, given the throughput $\rho = 0.525$ (21 tasks every 40 seconds), we have to build a K-pattern of length T_p such that $\rho = K/T_p$. The construction of such a pattern proceeds in two phases. First we decompose the solution of the linear program into a weighted sum of allocations $(\alpha_i, \mathcal{A}_i)$ (see Figure 4). Then we show how to simultaneously schedule these allocations without resource conflict. Theorem 2 proves that this last step is possible in a more general context. Nevertheless, we present in this section the actual reconstruction of the pattern on a simple example, for some readers may have (legitimately) skipped the proof of Theorem 2.

The main motivation to consider steady-state scheduling is that several distinct allocations are interleaved to squeeze the most out of the platform. In the simple case where G_A only contains a main task, an input task T_{begin} and an output task T_{end}, the decomposition into allocations is computed by peeling off the communication graph: see Algorithm 1.

Algorithm 1 depth-searches G_A and greedily select processors capable of executing the tasks. Conservation laws ensure that the desired allocation will always be found: when a task is consumed, it produces an output file which either is used locally, either is forwarded to another processor. Once we have a valid allocation, we determine its weight by taking the minimum of the quantities $cons(P_i, T_k)$ and $sent(P_i \to P_j, e_{k,l})$ involved in the allocation. We subtract this weight to these quantities, and we get updated variables still



(a) Application graph

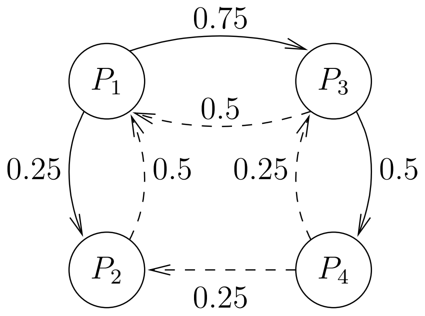


(b) Platform graph: input files $e_{begin,1}$ originally are on P_1 and output files $e_{1,end}$ have to be gathered on P_1

Figure 2: A simple example for reconstructing the schedule

	$\alpha(P_i, T_1)$	$cons(P_i, T_1)$
P_1	100%	0.025
P_2	100%	0.125
P_3	100%	0.125
P_4	100%	0.250
Total	21 tasks every 40 seconds	

(a) Computation times



(b) Communication times ($s(P_i \rightarrow P_j, e_{k,l})$)

Figure 3: Solution of the linear program : the platform graph is annotated with non-zero values of $s(P_i \rightarrow P_j, e_{k,l})$

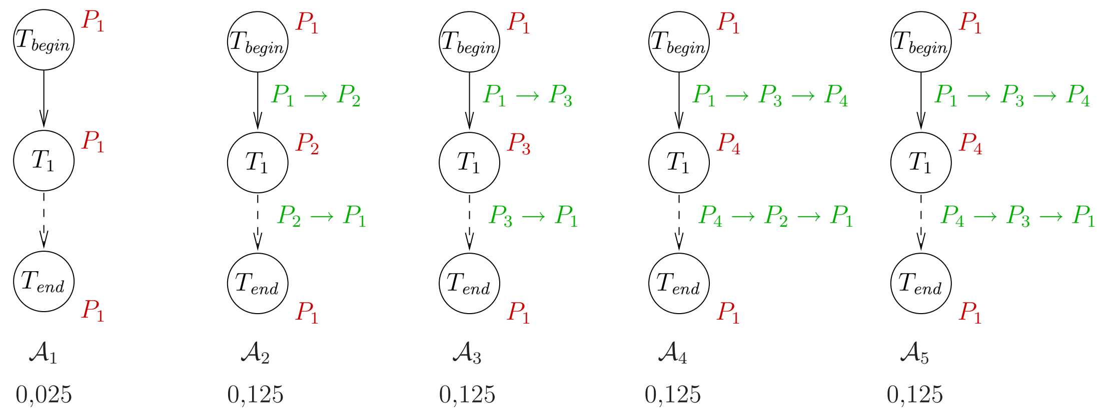
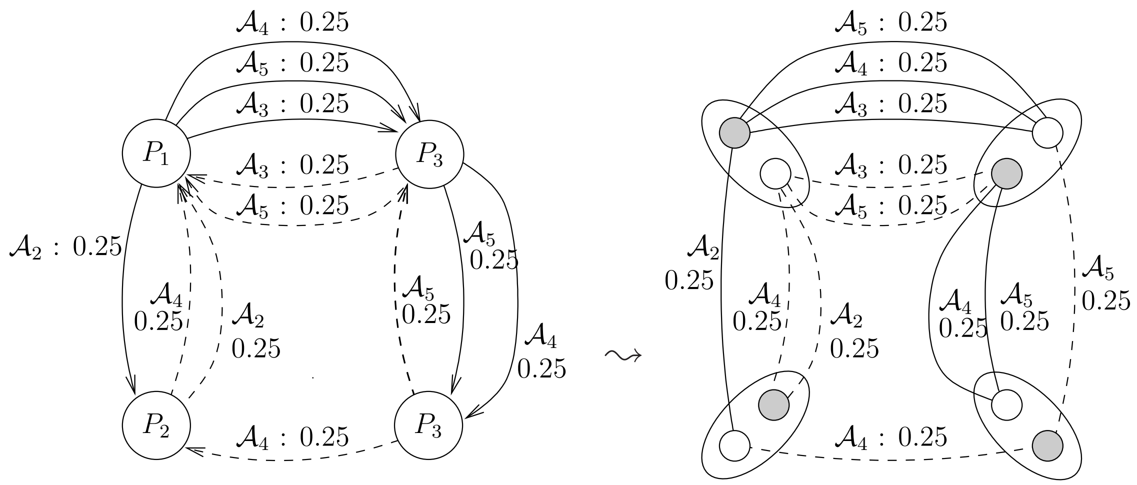


Figure 4: Decomposition of the solution of the linear program into 5 allocations $\mathcal{A}_1,\ldots,\mathcal{A}_5$. Each allocation contributes to a certain amount to the steady-state regime, and the sum of these contributions $(0.025 + 0.125 + 0.125 + 0.125 + 0.125)$ is equal to the total throughput total $(0.525 = \frac{21}{40})$.

FIND_AN_ALLOCATION()

Choose P_i such that $cons(P_i, T_{begin}) = 0$
$\pi(T_{begin}) \leftarrow P_i$.
$src \leftarrow i$
$\gamma_1 \leftarrow \emptyset$
while $cons(P_{src}, T_1) = 0$ **do**
    Choose P_j such that $sent(P_{src} \to P_j, e_{begin,1}) > 0$
    $\gamma_1 \leftarrow \gamma_1 \cup (P_{src} \to P_j)$
    $src \leftarrow j$
$\sigma(e_{begin,1}) \leftarrow \gamma_1$
$\pi(T_1) \leftarrow P_{src}$
$\gamma_2 \leftarrow \emptyset$
while $cons(P_{src}, T_{end}) = 0$ **do**
    Choose P_j such that $sent(P_{src} \to P_j, e_{1,end}) > 0$
    $\gamma_2 \leftarrow \gamma_2 \cup (P_{src} \to P_j)$
    $src \leftarrow j$
$\sigma(e_{1,end}) \leftarrow \gamma_2$
$\pi(T_{end}) \leftarrow P_{src}$
$\alpha \leftarrow \min\big(cons(P_{begin}, T_{\pi(T_{begin})}), cons(P_1, T_{\pi(T_1)}), cons(P_{end}, T_{\pi(T_{end})}),$
$\{sent(P_i \to P_j, e_{begin,1}) | P_i \to P_j \in \sigma(e_{begin,1})\},$
$\{sent(P_i \to P_j, e_{1,end}) | P_i \to P_j \in \sigma(e_{1,end})\}\big)$
return(α, π, σ)

Algorithm 1: Algorithm for extracting an allocation



(a) Solution of the linear program

(b) Bipartite graph

Figure 5: Communication graph and associated bipartite graph.

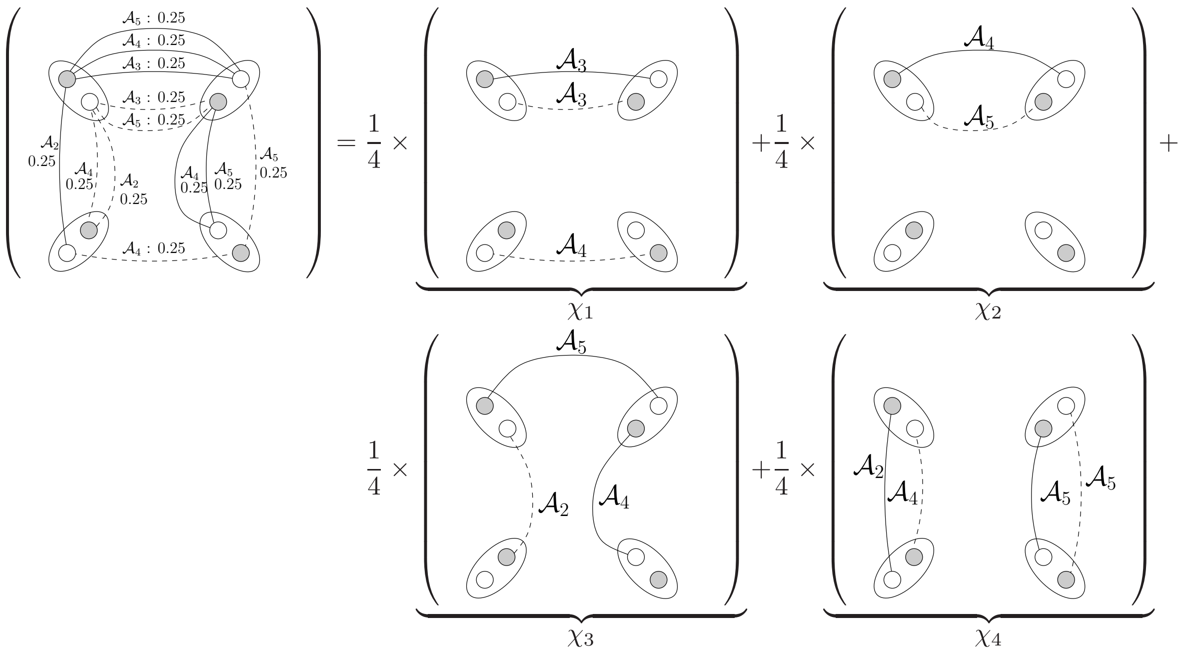
satisfying to all constraints (10): we start again searching for another allocation, until no more task can be consumed.

Once we have the allocations, we can directly use Theorem 2 to reconstruct the periodic steady-state schedule. However we illustrate the reconstruction on this example. The rest of this section can be skipped by the reader who has followed the proof of Theorem 2.

From the weighted allocations we build the *communication graph* shown in Figure 5(a). Each node is a processor. Solid edges represent transfers of type $e_{begin,1}$, while dashed edges represent transfers of type $e_{1,end}$. Each edge is labeled with an allocation number and a weight which represents the time fraction spent communicating the corresponding file. For instance, the solid edge $(\mathcal{A}_3, 0.25)$ from P_1 to P_2 means that every time-unit, one fourth of the time (weight 0.25) is spent forwarding from P_1 to P_2 files of type $e_{begin,1}$ and used for $\mathcal{A}_3$. There may be several edges between processor pairs.

To reconstruct a schedule from this description, we transform the communication graph into a weighted bipartite graph (see Figure 5(b)). We split each node into two nodes, one for the sends (in grey) and one for the receives (in white).

Because of the one-port model, there are at most two communications involving the same processor at a given time-step: one for sending, and one for receiving. The one-port constraints are enforced locally by Equations (5) and (6), but they have to be satisfied at the platform level. This is the role of the bipartite graph: we have to extract communications that can take place simultaneously, without violating the one-port constraints. Any set of communications which corresponds to a matching in the bipartite graph will do the job: by definition of a matching, any sending processor will be involved at most once, and the same for any receiving processor. Therefore, the bipartite graph must be decomposed into a weighted sum of matchings such that the sum of the weights is not greater than 1 (in order to orchestrate all communications within a time-unit). Back to the example, we can have the following decomposition:



(11)

Here, χ denotes the characteristic function of the matching ($\chi(e) = 1$ iff edge e belongs to the matching). Such a decomposition always exist and can be obtained. Decomposing a bipartite graph into a weighted sum of matchings amounts to derive a weighted edge-coloring of the graph. The algorithm, although complicated, is well-known, and its complexity is $O(m^2)$, where m is the number of edges in the bipartite graph [34, chapitre 20]. The number of matchings is bounded by m. From the edge-coloring decomposition, it is easy to reconstruct the schedule (see Figure 6) because we now have a pattern that satisfies to all resource and one-port constraints.

To summarize, the design of the final schedule is conducted as follows:

1. Solve the linear program (10).

2. Decompose the solution into a sum of allocations and build the communication graph induced by the $s(P_i \to P_j, e_{k,l})$.

3. Transform the communication graph into a weighted bipartite graph B.

4. Decompose graph B into a weighted sum of matchings $B = \sum \alpha_c \chi_c$ such that $\sum \alpha_c \leqslant 1$.

5. Letting $\alpha(P_i, T_1) = \frac{a_i}{b_i}$ and $\alpha_c = \frac{p_c}{q_c}$, we define the period T_p as the least common multiple of the $b_i \times w_{i,k}$ and of the $q_c \times data_{k,l} \times c_{i,j}$ if the transfer of a file of type $e_{k,l}$ from P_i to P_j belongs to matching χ_c. As a consequence, an integer number of each task type is consumed during each period T_p (owing to the $\alpha(P_i, T_1)$) and an integer number of files is transferred during each of the time-intervals corresponding to the matchings. Back to the example, T_p is the least common multiple of $10, 2, 2, 1$ (for P_1, P_2, P_3, P_4) and of $4 \times 2 = 8$ for the matchings (here, transfer times and weights are the same for all matchings), leading to the value $T_p = 40$.

6. By construction, each matching χ_i is a set of compatible communications of files $e_{begin,1}$ or $e_{1,end}$ from a P_i to a P_j and corresponding to a given allocation. Hence we have built a K-pattern (here, $K = 1 + 5 + 5 + 5 + 5 = 21$) of length T_p where K/T_p is equal to the optimal throughput. Using the one-to-one correspondence between patterns and periodic schedules of Section 2.6, we derive the desired periodic schedule of optimal throughput.

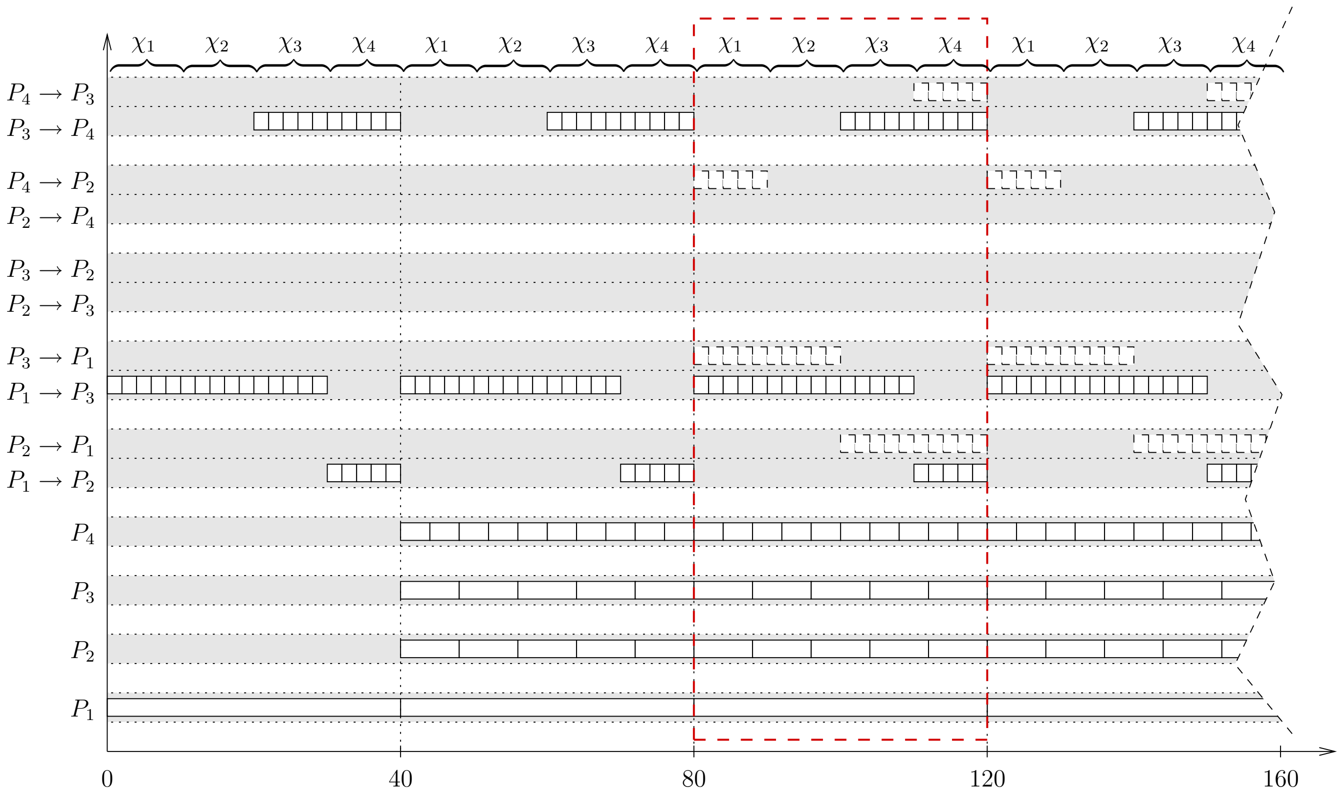


Figure 6: Schedule pattern achieving the optimal steady-state throughput

We point out that the description of the schedule has a size polynomial in the size of the problem input. Indeed, the number of variables in the linear program is $O(n^2p^2)$ (because of $s(P_i \to P_j, e_{k,l})$), so the bipartite graph has a size polynomial in n et p. The number of allocations is also bounded by n^2p^2 and the communication graph has at most $O(n^4p^4)$ edges. During the edge-coloring decomposition, the number of matchings is bounded by n^4p^4 and the description of the weighted sum is polynomial. The description of the schedule is thus polynomial, and the period length can be encoded in polynomial size too, as it is obtained as the least common multiple of the $b_i \times w_{i,k}$ and of $q_c \times data_{k,l} \times c_{i,j}$.

However, if the period length can be encoded in polynomial size, the period itself may not be polynomial. In other words, $\log T_p$ is polynomial in n and p but maybe T_p is not. Because we describe the operation of the processors by time-intervals corresponding to each allocation, we do have a polynomial-size description. It is important to realize that if we had described the operation of each resource at each time-step, we might have obtained a description of exponential size.

Apart from some periods in the very beginning (initialization) and and in the end (clean-up), the cyclic schedule achieves a perfect use of the resources. We show in Section 3.3 that the cyclic schedule is asymptotically optimal: in T time-steps, any schedule cannot execute more than a constant number of extra tasks than the cyclic schedule, and this constant is independent of T.

3.3 Asymptotic optimality

Let $opt(G_P, K)$ denote the optimal number of tasks that can be computed on the platform G_P within K time-units. Note that *optimal* refers to any valid schedule, not only periodic ones. We let ρ be the optimal throughput, computed using the linear program 10.

Lemma 3. $opt(G_P, K) \leqslant \rho \times K$

Proof. Consider an optimal schedule. For each processor P_i, and each task type T_k, let $t_{i,k}(K)$ be the total number of tasks of type T_k that have been executed by P_i within the K time-units. Similarly, for each processor pair (P_i, P_j) in the platform graph, and for each edge $e_{k,l}$ in the application graph, let $t_{i,j,k,l}(K)$ be the total number of files of type $e_{k,l}$ tasks that have been forwarded by P_i to P_j within the K time-units. The following inequalities hold true:

- $\forall i$, $\sum_k t_{i,k}(K).w_{i,k} \leqslant K$ (time for P_i to process its tasks)
- $\forall i$, $\sum_{P_i \to P_j} \sum_{k,l} t_{i,j,k,l}(K).data_{k,l}.c_{i,j} \leqslant K$ (time for P_i to forward outgoing tasks in the one-port model)
- $\forall i$, $\sum_{P_j \to P_i} \sum_{k,l} t_{j,i,k,l}(K).data_{k,l}.c_{i,j} \leqslant K$ (time for P_i to receive incoming tasks in the one-port model)
- $\forall e_{k,l}$, $\sum_{P_j \to P_i} t_{j,i,k,l}(K) + t_{i,k}(K) = \sum_{P_i \to P_j} t_{i,j,k,l}(K) + t_{i,l}(K)$ (conservation equation holding for each edge type $e_{k,l}$)

Let $cons(P_i, T_k) = \frac{t_{i,k}(K)}{K}$, $sent(P_i \to P_j, e_{k,l}) = \frac{t_{i,j,k,l}(K)}{K}$. We also introduce $\alpha(P_i, T_k) = cons(P_i, T_k).w_{i,k}$ and $s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}).data_{k,l}.c_{i,j}$. All the equations of the linear program 10 hold, hence $\sum_{i=1}^{p} cons(P_i, T_{end}) \leqslant \rho$, the optimal value.

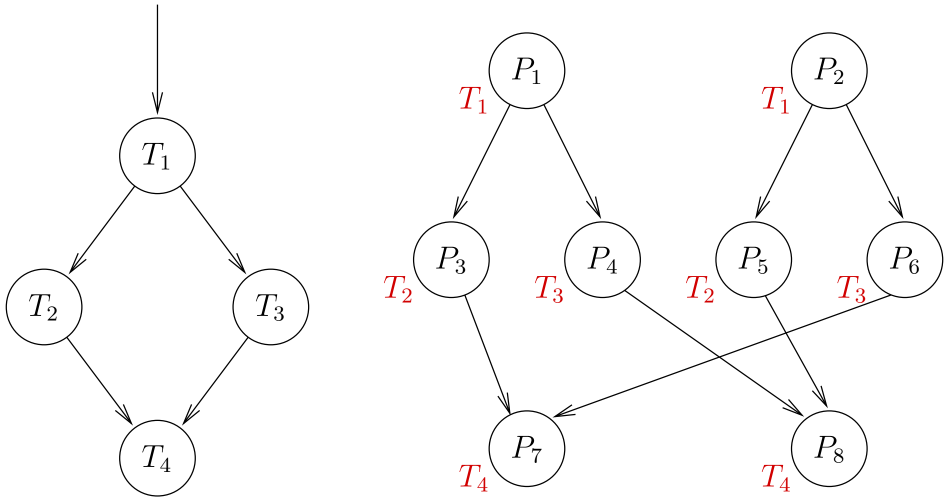
Going back to the original variables, we derive:

$$opt(G_P, K) = \sum_i t_{i,end}(K) \leqslant \rho \times K \qquad \blacksquare$$

Basically, Lemma 3 says that no scheduling can execute more tasks than the steady state scheduling. There remains to bound the loss due to the initialization and clean-up phases in the periodic schedule that has been reconstructed in the previous section. Consider the following (brute-force) approach (assume K is large enough):

- Solve the linear program 10: compute the maximal throughput ρ, compute all the values $\alpha(P_i, T_k)$, $cons(P_i, T_k)$, $s(P_i \to P_j, e_{k,l})$ and $sent(P_i \to P_j, e_{k,l})$. Determine the period T_p. For each processor P_i, determine $per_{i,k,l}$, the total number of files of type $e_{k,l}$ that it receives per period. Note that all these quantities are independent of K: they only depend upon the characteristics $w_{i,k}$, $c_{i,j}$, and $data_{k,l}$ of the platform and application graphs.
- Initialization: the master sends $per_{i,k,l}$ files of type $e_{k,l}$ to each processor P_i. To do so, the master generates (computes in place) as many tasks of each type as needed, and sends the files sequentially to the other processors. This requires I units of time, where I is a constant independent of K.
- Similarly, let J be the time needed by the following clean-up operation: each processor returns to the master all the files that it holds at the end of the last period, and the master completes the computation sequentially, generating the last copies of T_{end}. Again, J is a constant independent of K.
- Let $r = \lfloor \frac{K-I-J}{T_p} \rfloor$.
- Steady-state scheduling: during r periods of time T_p, operate the platform in steady-state, according to the solution of the linear program.
- Clean-up during the J last time-units: processors forward all their files to the master, which is responsible for terminating the computation. No processor (even the master) is active during the very last units ($K - I - J$ may not be evenly divisible by T_p).
- The number of tasks processed by this algorithm within K time-units is equal to $steady(G_P, K) = (r+1) \times T \times \rho$.

Clearly, the initialization and clean-up phases would be shortened for an actual implementation, using parallel routing and distributed computations. But on the theoretical side, we do not need to refine the previous bound, because it is sufficient to prove the following result:



(a) Application graph

(b) Platform graph: each processor is able to process only one task type

Figure 7: Counter-example

Theorem 3. *The previous scheduling algorithm based upon steady-state operation is asymptotically optimal:*

$$\lim_{K \to +\infty} \frac{steady(G_P, K)}{opt(G_P, K)} = 1.$$

Proof. Using Lemma 3, $opt(G_P, K) \leqslant \rho.K$. From the description of the algorithm, we have $steady(G_P, K) = ((r+1)T).\rho \geqslant (K - I - J).\rho$, hence the result because I, J, T and ρ are constants independent of K. ∎

4 Steady-state scheduling of a general application graph

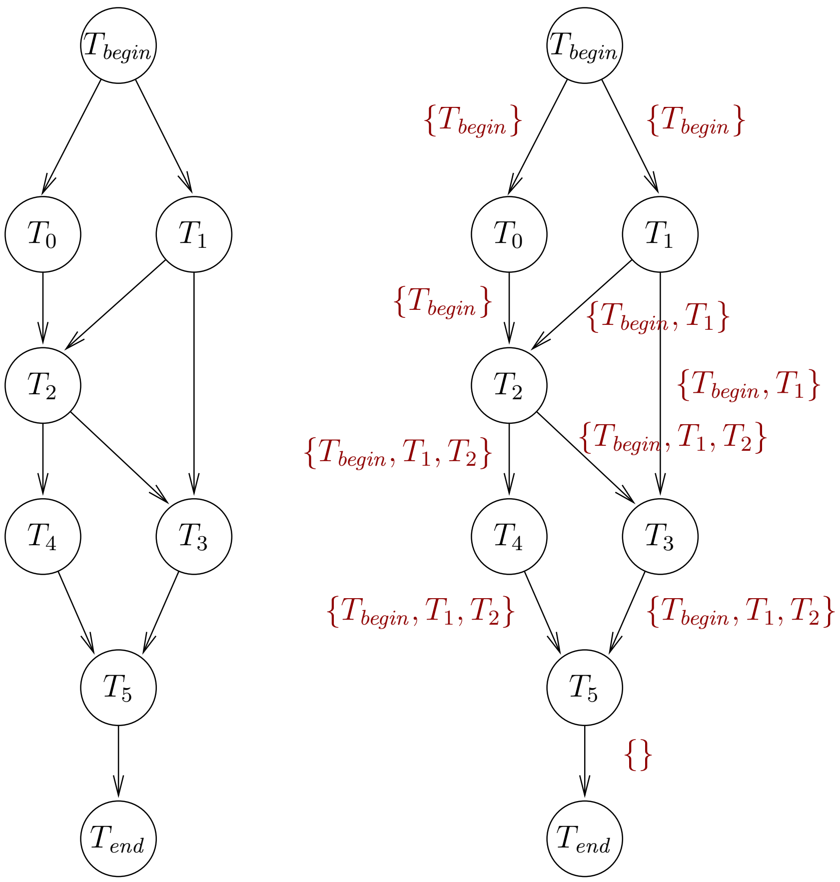
In this section we move from independent tasks to general application graphs, which can be arbitrary direct acyclic graphs (DAGs). As we will see, scheduling arbitrary DAGs turns out to be much more difficult than scheduling independent tasks.

4.1 Why are DAGs difficult?

Consider the problem of mapping the application graph depicted on Figure 7(a) onto the platform depicted on Figure 7(b). Processors are labeled with the task types that are able to execute. For instance, T_1 can only be executed by P_1 or P_2. Looking carefully at the figure, we check that no schedule is feasible. However, assuming that communication and communication times are all equal to one, if we had used the same equations for the linear program as before, we would have get an expected throughput of 2 application graphs per time-unit. The difficulty arises from the join node T_4 of the application graph. We need to merge data that corresponds to the same initial instance of the application graph. Therefore we need to keep track of the schedule of some ancestors to ensure that join parts of the DAG will be done correctly.

4.2 Adding constraints

To avoid the problem exposed in the previous section, we keep track of the schedule by adding some informations to each variable.



(a) Application graph

(b) Annotating the application graph: $e_{1,3}$ is labeled by T_{begin} and T_1 because of T_3 ; $e_{1,2}$ is labeled by T_{begin} and T_1 because of T_3 ; $e_{2,3}$ is labeled by T_{begin}, T_1 and T_2 because of T_3 and T_5 ; T_{end} is not labeled because the only access path to it goes through T_5 ;etc.

Figure 8: Computing constraints.

4.2.1 Definitions and algorithm

The variables $sent(P_i \to P_j, e_{k,l})$, $s(P_i \to P_j, e_{k,l})$, $\alpha(P_i, T_k)$ and $cons(P_i, T_k)$ will be annotated with a list of constraints L and will be written $sent(P_i \to P_j, e^L_{k,l})$, $s(P_i \to P_j, e^L_{k,l})$, $\alpha(P_i, T^L_k)$ and $cons(P_i, T^L_k)$. These constraint lists are the schedule of some ancestors, e.g. $\{T_{begin} \to P_1, T_1 \to P_2, T_3 \to P_2\}$. We now explain how to build these constraint lists.

Definition 14. *Given a dependency $e_{k,l}$ we define the set of* constraining tasks *of $e_{k,l}$ as the ancestors T_a of T_l for which there exists a T_d which is a descendant of T_a and which is not an ancestor of T_l.*

The *constraining tasks* of the $e_{k,l}$ are the tasks whose schedule is crucial to be memorized to ensure that join parts of the DAG will be done correctly. They can be constructed with Algorithm 2, which is illustrated on Figure 8.

We now define the constraint lists for the tasks T_k and the $e_{k,l}$. We distinguish between two types of constraints for a task T_k, depending whether these constraints have to be verified to process T_k (i.e. for all $e_{l,k}$) or whether all files $e_{k,l}$ have to respect these constraints.

Definition 15. *A constraint list for an edge $e_{k,l}$ is a mapping from $\{T_{k_1}, \ldots, T_{k_q}\}$ to $\{P_1, \ldots, P_p\}$, where*

```
Activate(T_u)
  Active[T_u] ← 1
  for all T_w s.a. T_u → T_w do
    if Active[T_w] ≠ 1 then
      ToActivate[T_w]
      inc(counter)
  dec(counter)
  ToActivate[T_u] ← 0

Remember(T_u,T_v)
  if counter>1 then
    for all T_w s.a. T_v → T_w do
      List[e_{v,w}] ← List[e_{v,w}] ∪ T_u

ComputeListe()
  for all e_{k,l} do
    List[e_{k,l}] ← ∅
  for all T_u do
    Counter ← 1
    Activate(T_u)
    Remember(T_u,T_u)
    while |ToActivate| > 0 and counter > 1 do
      To_Activate:
      for all T_v s.a. ToActivate[T_v]=1 do
        nb ← 0
        for all T_w s.a. T_w → T_v do
          if (there is a path from T_u to T_w) then
            if Active[T_w] then
              next To_Activate
            inc nb
        Activate(T_v)
        counter ← counter - nb + 1
        Remember(T_u,T_v)
```

Algorithm 2: Computing the constraining tasks.

$\{T_{k_1}, \ldots, T_{k_q}\}$ is set of constraining tasks of $e_{k,l}$. It is represented as a list of the form $\{T_{k_1} \rightarrow P_{i_1}, \ldots, T_{k_q} \rightarrow P_{i_q}\}$

Definition 16. *$Cnsts(e_{k,l}) = \{$ constraint list for $e_{k,l}$ $\}$*
$CnstsIn(T_k) = \{$ mapping from $\bigcup_{e_{l,k}}$(constraining tasks of $e_{l,k}$) to $\{P_1, \ldots, P_p\}$ $\}$
$CnstsOut(T_k) = \begin{cases} Cnsts(e_{k,l}) & \text{if there exists an } e_{k,l} \text{ in } E_A \\ \emptyset & \text{otherwise} \end{cases}$ *(by construction, for a given k, all the $e_{k,l}$ have the same list of constraints).*

The following definition enables to link constraint lists in $Cnsts(e_{k,l})$, $CnstsIn(T_k)$, and $CnstsOut(T_k)$.

Definition 17. *Two constraint lists L_1 and L_2 are* compatible *iff*

$$\forall (T_k \rightarrow P_i) \in L_1, \forall P_j \neq P_i, (T_k \rightarrow P_j) \notin L_2$$

The following two definitions simply help to build the equations of the linear program.

Definition 18. *A file $e_{k,l}$ respecting constraints $L \in Cnsts(e_{k,l})$ can be transferred from P_i to P_j iff $c_{i,j} \neq \infty$.*

Definition 19. *A task T_k can be processed on processor P_i under constraints $L \in CnstsOut(T_k)$ iff $w_{i,k} \neq \infty$ and if processing T_k on P_i does not violate one of the constraints of L (i.e. if there's not a $P_j \neq P_i$ such as $(T_k \rightarrow P_j) \in L$).*

4.2.2 Equations

- For each edge $e_{k,l} : T_k \rightarrow T_l$ in the task graph, for each processor pair (P_i, P_j) and each valid constraint list $L \in Cnsts(e_{k,l})$, we denote by $s(P_i \rightarrow P_j, e_{k,l}^L)$ the (average) fraction of time spent each time-unit by P_i to send to P_j data involved by the edge $e_{k,l}$ under constraints L. As usual $s(P_i \rightarrow P_j, e_{k,l}^L)$ is a nonnegative rational number. Let the (fractional) number of such files sent per time-unit be denoted as $sent(P_i \rightarrow P_j, e_{k,l}^L)$. We have the same kind of relation as before:

$$s(P_i \rightarrow P_j, e_{k,l}^L) = sent(P_i \rightarrow P_j, e_{k,l}^L) \times (data_{k,l} \times c_{i,j}) \tag{12}$$

  which states that the fraction of time spent transferring such files is equal to the number of files times the product of their size by the elemental transfer time of the communication link.

- For each task type $T_k \in V$, for each processor P_i and for each valid constraint list $L \in CnstsOut(T_k)$, we denote by $\alpha(P_i, T_k^L)$ the (average) fraction of time spent each time-unit by P_i to process tasks of type T_k fulfilling constraints L, and by $cons(P_i, T_k^L)$ the (fractional) number of tasks of type T_k fulfilling constraints L processed per time unit by processor P_i. We have the relation

$$\alpha(P_i, T_k^L) = cons(P_i, T_k^L) \times w_{i,k}. \tag{13}$$

  Before being processed on P_i, a task T_k has to be ready i.e. the files necessary to its processing have to be gathered on P_i. The constraint list of the input files (belonging to $CnstsIn(T_k)$) of a task and the constraint list of the output files (belonging to $CnstsOut(T_k)$) are generally different. It may shrink (e.g. T_5 on Figure 8(b)) or grow (e.g. T_1 on Figure 8(b)). That is why we distinguish the tasks that are ready to be processed under some constraints ($prod(P_i, T_k)$) from the one that have just been processed and have produced some output files ($cons(P_i, T_k)$). Therefore, we have the following equation linking $prod(P_i, T_k)$ and $cons(P_i, T_k)$:

$$cons(P_i, T_k^L) = \sum_{\substack{L_2 \in CnstsIn(T_k) \\ T_k \text{ can be processed on } P_i \text{ under constraints } L_2 \\ L \text{ and } L_2 \text{ are compatible}}} prod(P_i, T_k^{L_2}) \tag{14}$$

Activities during one time-unit All fractions of time spent by a processor to do something (either computing or communicating) must belong to the interval $[0,1]$, as they correspond to the average activity during one time unit:

$$\forall P_i, \forall T_k \in V, \forall L \in \mathit{CnstsOut}(T_k)\ 0 \leqslant \alpha(P_i, T_k^L) \leqslant 1 \tag{15}$$

$$\forall P_i, P_j, \forall e_{k,l} \in E, \forall L \in \mathit{Cnsts}(e_{k,l})\ 0 \leqslant s(P_i \to P_j, e_{k,l}^L) \leqslant 1 \tag{16}$$

One-port model for outgoing communications Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i, \sum_{\substack{P_j \in n(P_i)\\ e_{k,l} \in E\\ L \in \mathit{Cnsts}(e_{k,l})}} s(P_i \to P_j, e_{k,l}^L) \leqslant 1 \tag{17}$$

where $n(P_i)$ denotes the neighbors of P_i.

One-port model for incoming communications Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall P_i, \sum_{\substack{P_j \in n(P_i)\\ e_{k,l} \in E\\ L \in \mathit{Cnsts}(e_{k,l})}} s(P_j \to P_i, e_{k,l}^L) \leqslant 1 \tag{18}$$

Note that $s(P_j \to P_i, e_{k,l})$ is indeed equal to the fraction of time spent by P_i to receive from P_j files of type $e_{k,l}$.

Full overlap Because of the full overlap hypothesis, there is no further constraint on $\alpha(P_i, T_k^L)$ except that

$$\forall P_i, \sum_{\substack{T_k \in V\\ L \in \mathit{CnstsOut}(T_k)}} \alpha(P_i, T_k^L) \leqslant 1 \tag{19}$$

4.2.3 Conservation laws and linear program

The last constraints deal with *conservation laws*. Consider a given processor P_i, and a given edge $e_{k,l}^L$ in the application graph annotated with the constraint list L. During each time unit, P_i receives from its neighbors a given number of files of type $e_{k,l}^L$: P_i receives exactly $\sum_{P_j \in n(P_i)} \mathit{sent}(P_j \to P_i, e_{k,l}^L)$ such files. Processor P_i itself executes some tasks T_k, namely $\mathit{cons}(P_i, T_k^L)$ tasks T_k^L, thereby generating as many new files of type $e_{k,l}^L$.

What does happen to these files? Some are sent to the neighbors of P_i, and some will be used to produce some ready $T_l^{L_2}$ (with L_2 compatible with L) that are going to be consumed by P_i. We derive the equation:

$$\begin{aligned}
&\forall P_i, \forall e_{k,l} \in E : T_k \to T_l, \forall L \in \mathit{Cnsts}(e_{k,l})\\
&\qquad \sum_{P_j \in n(P_i)} \mathit{sent}(P_j \to P_i, e_{k,l}^L) + \mathit{cons}(P_i, T_k^L) =\\
&\qquad\qquad \sum_{P_j \in n(P_i)} \mathit{sent}(P_i \to P_j, e_{k,l}^L) + \sum_{\substack{L_2 \in \mathit{CnstsIn}(T_l)\\ L \text{ and } L_2 \text{ compatible}}} \mathit{prod}(P_i, T_l^{L_2})
\end{aligned} \tag{20}$$

Altogether, we derive the following linear program, which resembles that of Section 3.1.4:

$$
\begin{array}{l}
\text{MAXIMIZE } \rho = \sum_{i=1}^{p} cons(P_i, T_{end}^{\{\}}),\\
\text{UNDER THE CONSSTRAINTS}\\
\left\{
\begin{array}{ll}
(21a) & s(P_i \rightarrow P_j, e_{k,l}^L) = sent(P_i \rightarrow P_j, e_{k,l}^L) \times (data_{k,l} \times c_{i,j})\\
(21b) & \alpha(P_i, T_k^L) = cons(P_i, T_k^L) \times w_{i,k}\\
(21c) & cons(P_i, T_k^L) = \displaystyle\sum_{\substack{L_2 \in CnstsIn(T_k)\\ T_k \text{ may be processed by } P_i \text{ and satisfies to } L_2\\ L \text{ et } L_2 \text{ are compatible}}} prod(P_i, T_k^{L_2})\\
(21d) & \forall P_i, \displaystyle\sum_{\substack{P_i \rightarrow P_j\\ e_{k,l} \in E_A\\ L \in Cnsts(e_{k,l})}} s(P_i \rightarrow P_j, e_{k,l}^L) \leqslant 1\\
(21e) & \forall P_i, \displaystyle\sum_{\substack{P_j \rightarrow P_i\\ e_{k,l} \in E_A\\ L \in Cnsts(e_{k,l})}} s(P_j \rightarrow P_i, e_{k,l}^L) \leqslant 1\\
(21f) & \forall P_i, \displaystyle\sum_{\substack{T_k \in V_A\\ L \in CnstsOut(T_k)}} \alpha(P_i, T_k^L) \leqslant 1\\
(21g) & \forall P_i, \forall e_{k,l} \in E_A : T_k \rightarrow T_l, \forall L \in Cnsts(e_{k,l})\\
& \qquad \displaystyle\sum_{P_j \rightarrow P_i} sent(P_j \rightarrow P_i, e_{k,l}^L) + cons(P_i, T_k^L) =\\
& \displaystyle\sum_{P_i \rightarrow P_j} sent(P_i \rightarrow P_j, e_{k,l}^L) + \sum_{\substack{L_2 \in CnstsIn(T_l)\\ L \text{ and } L_2 \text{ compatible}}} prod(P_i, T_l^{L_2})
\end{array}
\right.
\end{array}
\tag{21}
$$

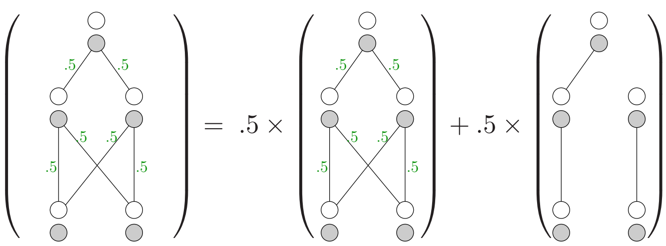
4.3 Reconstruction on an effective schedule

Section 4.3.1 focuses on a simple example to explain why the reconstruction of the schedule is more difficult than for independent tasks. Section 4.3.2 presents a method to decompose the solution of the linear program into a weighted sum of allocations, and Section 4.3.3 shows how to mix those allocations to get an effective schedule.

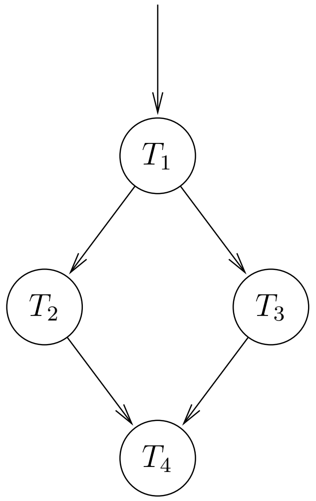
4.3.1 Why are DAGs more difficult (continued)

In the same way as computing the optimal throughput of a DAG on a general platform turns out to be much more difficult than for independent tasks, reconstructing a schedule is a little bit more tricky, even with the constraints introduced in Section 4.2.

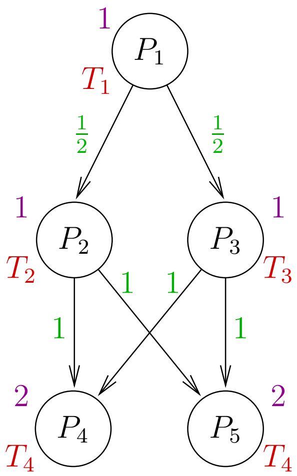
Consider the example of Figure 9. Each processor in the platform of Figure 9(b)) can only execute one task type; this task type and its execution time are indicated close to the processors. Each dependence file has unit cost, so that the weight of the edges in the platform graph represent the time needed to communicate the file. Using the linear program (21), we compute $\rho = 1$, i.e. one application graph per time-unit. Using the same technique as in Section 2, we transform the communication graph into a bipartite graph and we get the following decomposition:



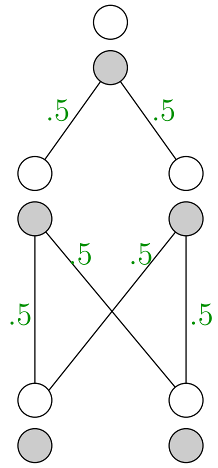
(22)



(a) Application graph. All weights are identical



(b) Platform graph



(c) Bipartite graph transformation

Figure 9: Difficulty of schedule reconstruction

Nevertheless, join parts have to be treated carefully. Figure 10 depicts two allocations. The first one is constructed by using a breadth-first descent of the application graph and is incorrect because files corresponding to $e_{2,4}$ and $e_{3,4}$ for the first instance of the application graph are sent to different processors (resp. P_4 and P_5). This incompatibility could be avoided by adding some constraints when designing the linear program (by remembering fork *and* join parts). Fortunately, correct allocations can be built traversing the application graph backwards (see Figure 10).

4.3.2 Algorithm for decomposing the solution into allocations

The platform that we use to illustrate our algorithm in this section is depicted in Figure 2(b) and the application graph is depicted on Figure 11.

The rationale of the approach is that mixing different allocations is likely to achieve an even better throughput that using a single one (which is furthermore difficult to find). This section explains how to decompose the solution of the linear program into a weighted sum of allocations. This is achieved by annotating the application graph with the non-zero values of $\mathit{cons}(P_i, T_k^L)$, $\mathit{prod}(P_i, T_k^L)$ and $\mathit{sent}(P_i \to P_j, e_{k,l}^L)$ (see Figure 12). The process is much easier when introducing $\mathit{sent}(P_i \to P_i, e_{k,l}^L)$, the amount of $e_{k,l}^L$ that are produced in place, that are not transferred to another processor and that stay in place for another computation. Hence, we have:

$$\forall P_i, \forall e_{k,l} \in E : T_k \to T_l, \forall L \in \mathit{Cnsts}(e_{k,l}) :$$

$$\mathit{sent}(P_i \to P_i, e_{k,l}^L) = \sum_{\substack{L_2 \,\in\, \mathit{CnstsIn}(T_l) \\ T_l \text{ can be processed on } P_i \text{ under constraints } L_2 \\ L \text{ and } L_2 \text{ are compatible}}} \mathit{prod}(P_i, T_l^{L_2}) - \sum_{\substack{L_1 \,\in\, \mathit{CnstsOut}(T_k) \\ L \text{ and } L_1 \text{ are compatible}}} \mathit{sent}(P_j \to P_i, e_{k,l}^{L_1}) \quad (23)$$

As explained in Section 4.3.1, we need to traverse the application graph backwards. Algorithm 3 builds a valid allocation from the solution of the linear program. The weight of this allocation, i.e. its throughput, is then equal to the minimum values over the $\mathit{cons}(P_i, T_k^L)$, $\mathit{prod}(P_i, T_k^L)$ and $\mathit{sent}(P_i \to P_j, e_{k,l}^L)$ involved in it. The decomposition into a weighted sum of allocations then simply consists in finding an allocation,

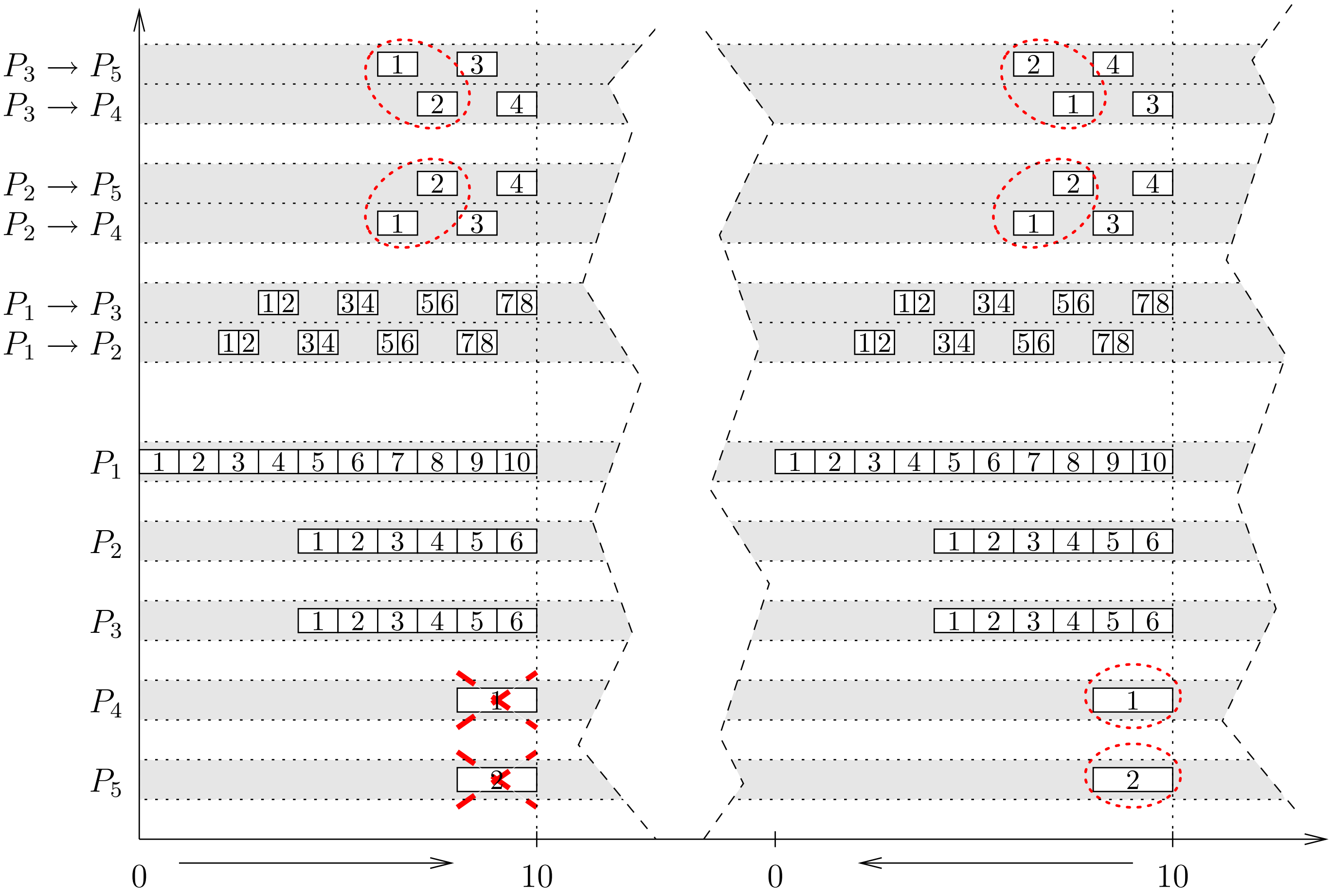


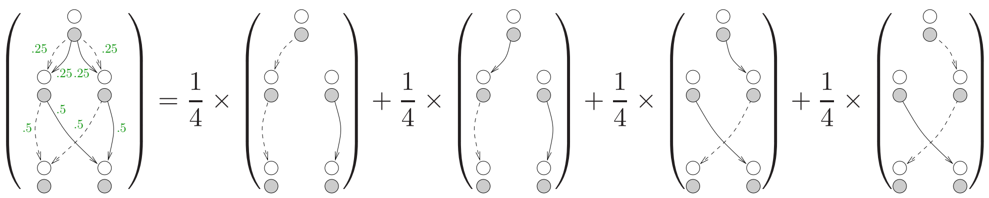
Figure 10: Effective schedules deduced from the decomposition of the bipartite graph

evaluating its weight and subtracting it to the solution of the linear program, until $cons(P_i, T_{end}) = 0$ for all P_i. The decomposition of the solution depicted Figure 12 is made of 10 different allocations. Figure 13 depicts the main two allocations (those with the largest weights).

4.3.3 Edge-coloring to ensure allocation compatibility

Once we have the allocations, we reconstruct the schedule as before, using Theorem 2. The approach is quite similar to that of Section 3.2.

Going back to the example of Section 4.3.1, with the platform graph depicted in Figure 9(b), the optimal throughput is obtained by mixing the two allocations depicted on Figure 14. The bipartite communication graph is shown in Figure 15. We obtain the decomposition:



(24)

Therefore, we are able to compute a schedule achieving the optimal throughput, just like we did in Section 3.2.

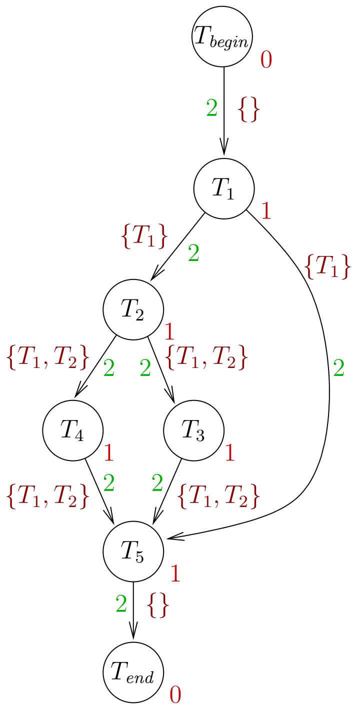


Figure 11: Not so simple an application graph.

4.4 Cost of the approach

If we denote by n the number of tasks in the application graph and by p the number of nodes in the platform graph, the number of variables involved in the linear program of Section 4.2.2 may be proportional to $p^2n^2p^n$. Indeed, when dealing with $sent(P_i \to P_j, e^L_{k,l})$, L being a list of constraints (i.e. an application from $\{T_{k_1}, \ldots, T_{k_q}\}$ to $\{P_1, \ldots, P_p\}$), the number of possible constraint list may be equal to p^n. This situation may happen on graphs with cascades of forking and very late joining. For example in the graph depicted on Figure 16, all the gray tasks need to be memorized to ensure a correct reconstruction of the double-circled one. On the contrary, on Figure 17, there is at most one task in the constraint list.

Definition 20. *The dependency depth is the maximum number of constraining tasks of the* $e_{k,l}$.

Theorem 4. *Let* $d \in \mathbb{N}$. *For all application graphs* (G_a, V_a) *whose dependency depth is bounded by* d, *and for all platform graphs* (G_p, V_p), *it is possible in polynomial time (i) to compute the optimal steady-state throughput and (ii) to reconstruct the periodic schedule that achieves this throughput.*

Proof. All the algorithms described in Section 4.3 are polynomial and their inputs are of size bounded by $p^2n^2p^d$. ■

The conclusion of this section can be either optimistic or pessimistic. The good news is that the approach is polynomial for a large collection of application graphs, such as series of fork-joins. The bad news is that our approach has an exponential cost for some application graphs, those with a large dependency depth (such as 2D meshes). In fact, unless P=NP, there is no polynomial algorithm in the general case: this important NP-hardness result is shown in Section 5.

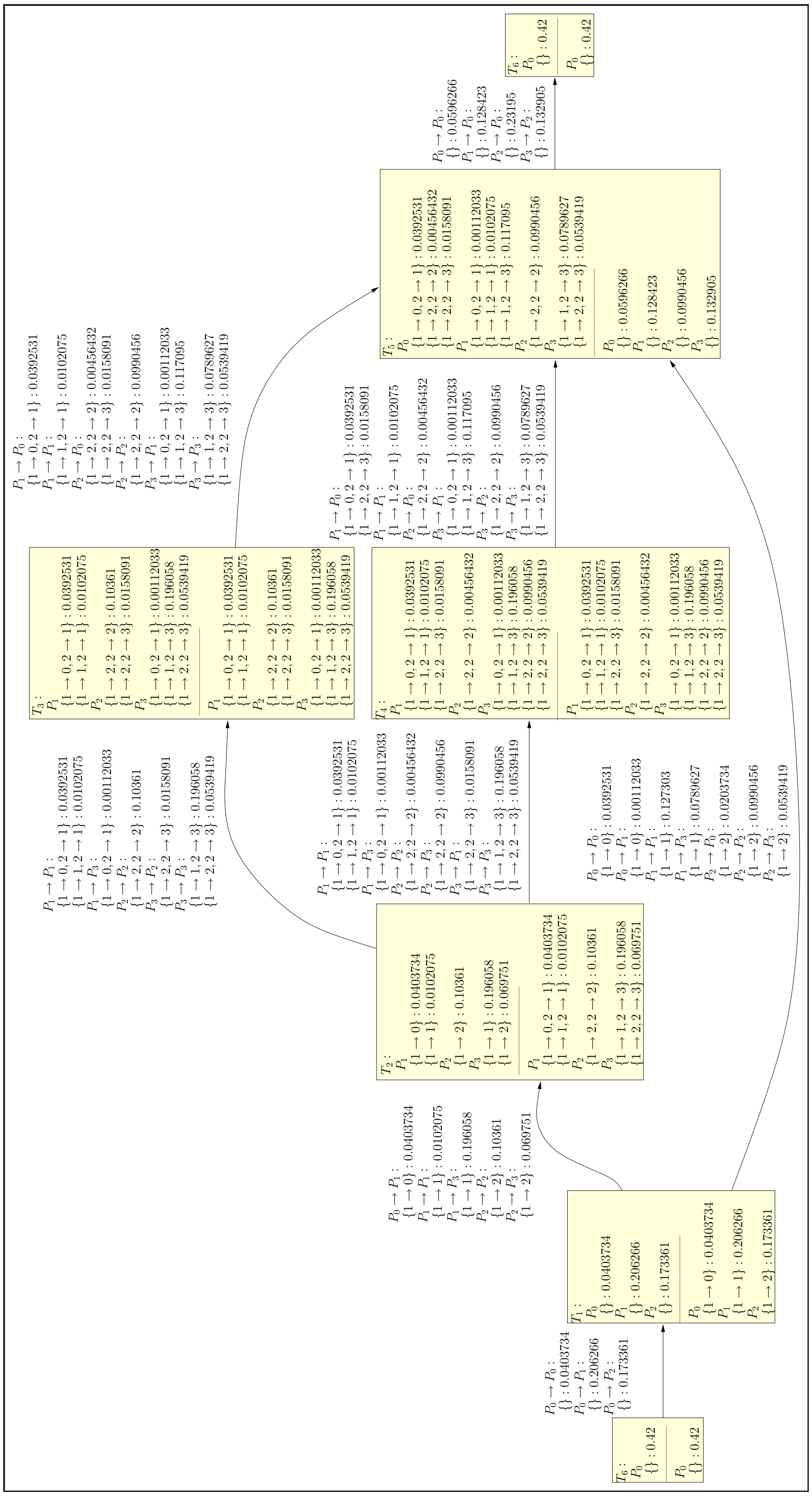
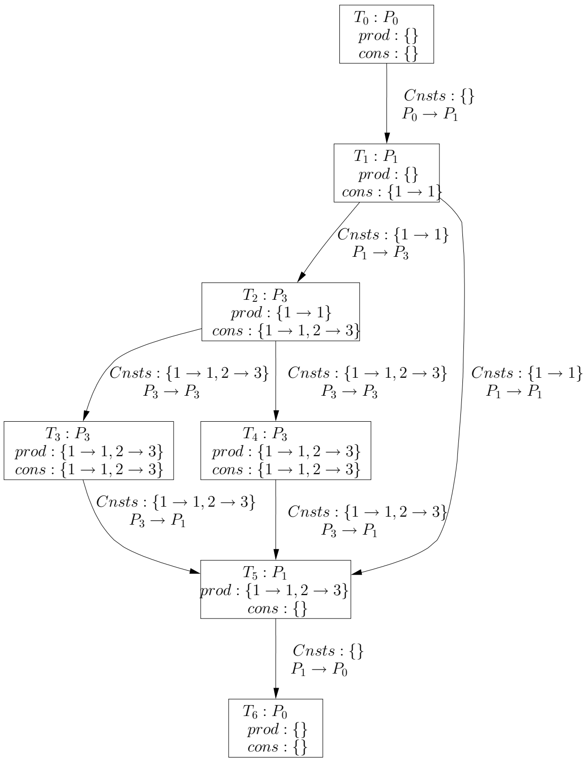
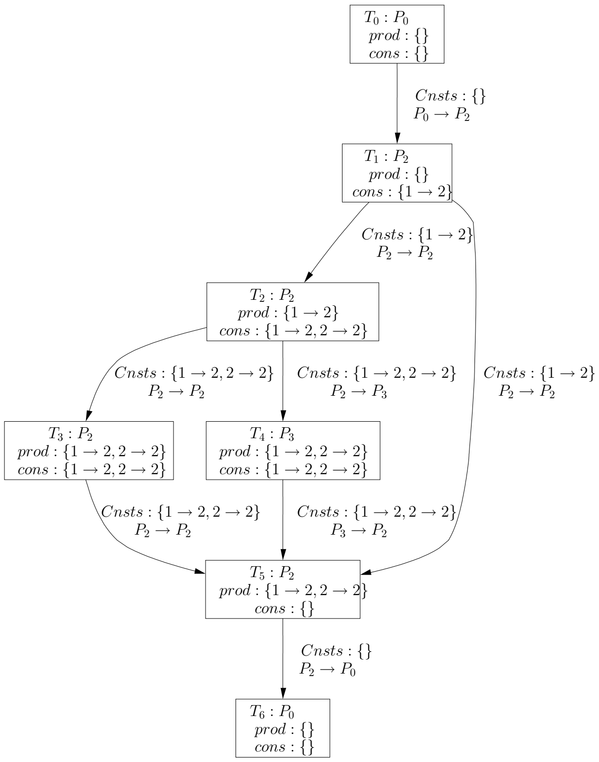


Figure 12: Solution given by the linear program : the application graph is annotated with the non-zero values of $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$, $sent(P_i \to P_j, e_{k,l}^L)$

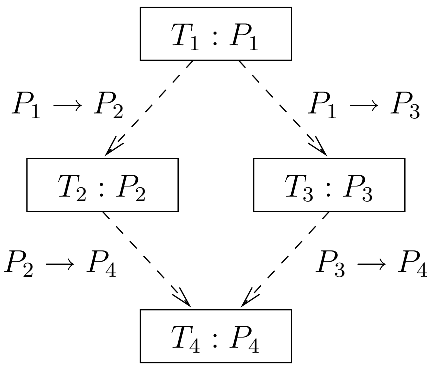


(a) Weight : 0.117095 (27.88%)

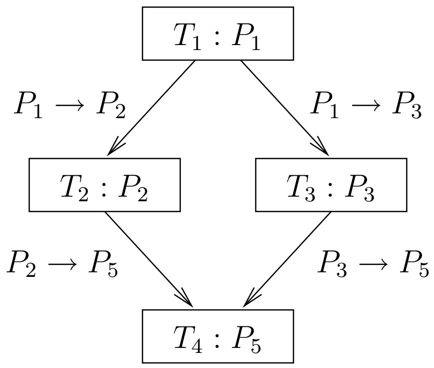


(b) Weight : 0.0990454 (23.58%)

Figure 13: Two main allocations obtained when decomposing the solution depicted Figure 12



(a) Weight : 0.5 (50.0%)



(b) Weight : 0.5 (50.0%)

Figure 14: Two allocations obtained when decomposing the optimal solution for the application graph of Figure 9(a) and the platform graph of Figure 9(b). For the sake of clarity, constraint lists are not depicted because they are useless on this particular example.

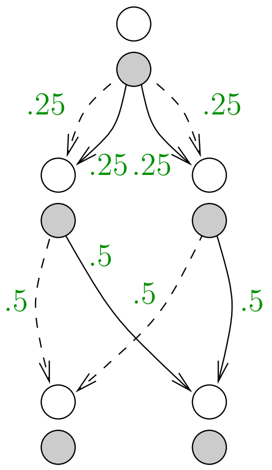


Figure 15: Bipartite graph associated to the platform graph depicted Figure 9(b) and to the allocations depicted on Figure 14. Each edge goes from a processor P_i to another processor P_j and is associated to an $e_{k,l}$ and a particular allocation. The weight of the edges is the fraction of time needed to transfer this file within this schedule from P_i to P_j.

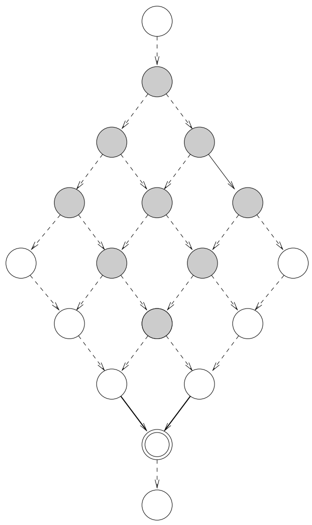


Figure 16: A 2D-mesh application graph. All the gray tasks need to be memorized to ensure a correct reconstruction of the double-circled one.

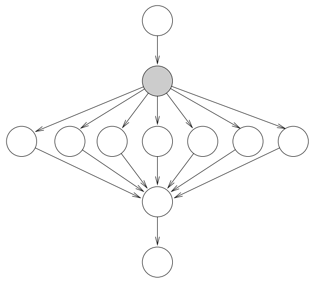


Figure 17: A fork application graph. Only the gray task needs to be memorized to ensure a correct reconstruction of the other tasks.

FIND_A_SCHEDULE()
$to\_activate \leftarrow \{T_{end}\}$
$CnstsCons(T_{end}) \leftarrow \emptyset$
$P(T_{end}) \leftarrow$ a P_i s.a. $prod(P_i, T_{end}^{\emptyset}) > 0$
while $to\_activate \neq \emptyset$ **do**
  $l \leftarrow$ POP($to\_activate$)
  $i \leftarrow P(T_l)$
  $L \leftarrow CnstsCons(T_l)$
  Let L_1 s.a. $prod(P_i, T_l^{L_1}) > 0$ and L_1 compatible with L
  $CnstsProd(T_l) \leftarrow L_1$
  if $T_l \neq T_{begin}$ **then**
    for all T_k s.a. $T_k \rightarrow T_l$ **do**
      Let L_2 and j s.a. $sent(P_j \rightarrow P_i, e_{k,l}^{L_2})$ and L_2 compatible with L_1
      $Cnsts(e_{k,l}) \leftarrow L_2$
      $CnstsCons(T_k) \leftarrow L_2$
      $transfer(e_{k,l}) \leftarrow \{P_j \rightarrow P_i\}$
      $src \leftarrow j$
      if $P_i \neq P_j$ and $prod(P_j, T_k^{L_2}) = 0$ **then**
        $dst \leftarrow j$
        repeat
          let $P_{src} \neq P_j$ s.a. $sent(P_{src} \rightarrow P_{dst}, e_{k,l}^{L_2}) > 0$
          $to\_activate \leftarrow to\_activate \cup \{P_{src} \rightarrow P_{dst}\}$
        until $prod(P_{src}, T_k^{L_2}) > 0$
      $P(T_k) \leftarrow P_{src}$

Algorithm 3: Algorithm for building an allocation

4.5 Hints for an actual implementation

4.5.1 Approximating the values

Let $S_1, \ldots, S_k$ be the allocations computed in Section 4.3.2 and $\alpha_1 = \frac{p_1}{T_p}, \ldots, \alpha_k = \frac{p_k}{T_p}$ be the throughputs of these allocations. If we denote by α_{opt} the optimal steady-state throughput, we have

$$\alpha_{opt} = \sum_{i=1}^{q} \alpha_i. \tag{25}$$

T_p may be very large and therefore impracticable: rounding the weights of the allocations may be necessary. Let us compute the throughput $\alpha(T)$ that can be achieved by simply rounding the α_i over a time period T. If we denote by $r_i(T)$ the number of DAGs that can be processed in steady-state using allocation S_i during a period T, we have :

$$r_i(T) = \lfloor \alpha_i T \rfloor \tag{26}$$

Note that, by using floor rounding, the equations of Section 4.2.2 still hold true and therefore lead to an effective schedule. We have the following equations:

$$\alpha_{opt} \geqslant \alpha(T) = \sum_{i=1}^{q} \frac{r_i(T)}{T} \geqslant \sum_{i=1}^{q} \frac{\alpha_i T - 1}{T} \geqslant \alpha_{opt} - \frac{q}{T} \tag{27}$$

We have proven the following result:

Proposition 1. *We can derive a steady-state operation for periods of arbitrary length, whose throughput converges to the optimal solution as the period size increases.*

4.5.2 Dynamic algorithm on general platforms

The algorithm presented in Section 4.3.3 to ensure the compatibility of the allocations requires a global clock and a global synchronization mechanism. A nice practical alternative is to use the 1D dynamic load balancing algorithm presented in [13] to decide on the fly which allocation should be used.

Let $S_1, \dots, S_k$ be the allocations computed in Section 4.3.2 and $\alpha_1, \dots, \alpha_k$ be the throughput of these allocations. We use the following dynamic programming algorithm:

```
DYNAMIC_ASSIGNMENT(α_1,...,α_p,B)
C = (c_1,...,c_p) = (0,...,0)
for b=1..B do
    i ← argmin_{1⩽j⩽p}((c_j + 1)/α_j)
    A(b) ← i
    c_i ← c_i + 1
return(A)
```

Algorithm 4: Dynamic programming algorithm for the optimal assignment of B independent identical chunks on p heterogeneous processors of relative speeds $\alpha_1, \dots, \alpha_p$.

For each value of $b \leqslant B$, let $C^{(b)} = (c_1^{(b)}, \dots, c_p^{(b)})$ denote the assignment of the first $b = \sum_{i=1}^{p} c_i$ chunks computed by the algorithm. This assignment is such as $\max \frac{c_i}{\alpha_i}$ is minimized [13]. Therefore, when allocating the allocations using the assignment A built with this algorithm, we respect the proportion of the weights of the allocations in the best possible way. Using such an approach on a real platform where load variations may occur, should lead to very good results while reducing the number of pending tasks on each processor. Such an approach has already be used for scheduling independent tasks on tree-shaped platforms [15].

5 Complexity results

In this section, we derive some complexity results for the problem of maximizing the throughput when mapping an application graph $G_A = (V_A, E_A)$ onto a given platform graph $G_P = (V_P, E_P)$. First we define a restricted version of the problem, whose solutions can be verified in polynomial time. We will show the NP-completeness of the restricted version. Before that, we show that we do not lose anything by sticking to the restricted version: if the general problem admits a solution of given throughput, so does the restricted version. Altogether, these results fully demonstrate the difficulty of the general problem.

Recall that $data_{k,l}$ denotes the volume of communications generated by task T_k for task T_l, for any edge $e_{k,l} \in E_A$, that $c_{i,j}$ denotes the time to transfer an elementary communication from P_i to P_j, for any edge $(P_i, P_j) \in E_P$, and that $w_{i,k}$ denotes the time to process task $T_k \in V_A$ on processor $P_i \in V_P$. The target decision problems can be stated as follows:

Definition 21 (GRAPH-THROUGHPUT(G_A, G_P, ρ)). *: Given a platform graph G_P, an application graph G_A and a rational bound for the throughput ρ, does there exist a periodic schedule whose throughput is at least ρ?*

However, we need a version where the solution can be verified in polynomial time:

Definition 22 (COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ)). *: Given a platform graph G_P, an application graph G_A and a rational bound for the throughput ρ, does there exist a periodic schedule consisting of at most $k \leqslant 3|V_P|$ allocations $\mathcal{A}_1, \dots, \mathcal{A}_k$, where the weight α_i is the average number of graphs processed by the allocation $\mathcal{A}_i$ within one time unit, $\alpha_i = \frac{a_i}{b_i}$, and a_i and b_i are integers such that*

$$\forall i, \quad \log a_i + \log b_i \leqslant 6|V_P|(2 + \log(|V_P|) + \log(M)),$$

$$\textit{where } M = \max(1, |V_A| \max w_{i,k}, |V_P||E_A| \max c_{i,j} \max data_{k,l})$$

and such that the throughput is at least ρ:

$$\sum \alpha_i \geqslant \rho \ ?$$

In the latter definition, we restrict the search to solutions where a bounded number ($k \leqslant 3|V_P|$) of allocations is used, whose weights can be expressed in a compact way ($\alpha_i = \frac{a_i}{b_i}$, where a_i and b_i are integers such that $\log a_i + \log b_i \leqslant 6|V_P|(\log(|V_P|) + \log(M))$. This restriction is necessary in order to keep the problem in the class NP, since an optimal solution may have a size exponential in the size of the initial data: indeed, from any periodic solution with period T, we can trivially build another solution, achieving the same throughput, with period $r \cdot T$, for any integer r. However, the following theorem asserts that this restriction on the size of the solution does not affect the optimal throughput:

Theorem 5. *Given a weighted application graph G_A and a weighted platform graph G_P, if there exists a periodic schedule to GRAPH-THROUGHPUT that achieves a throughput ρ, then there also exists a solution of COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ).*

Proof. In order to prove this result, we first derive a set of constraints that will be satisfied by any periodic solution to the GRAPH-THROUGHPUT problem. Let us denote by $\mathcal{A}$ the set of all possible allocations. There may be an exponential number of such allocations (with respect to the size of the application and platform graphs), but the number of allocations is nevertheless finite, since it consists in associating a given processor to any task, and a given path in the platform graph (of size at most $|V_P|$ since cycles are clearly useless) to any dependence in the application graph. A solution of the GRAPH-THROUGHPUT problem is then a set of weighted allocations $\{(\mathcal{A}_1, \alpha_1), \ldots, (\mathcal{A}_r, \alpha_r)\}$: here the weight α_m is the number of times per time-unit where allocation $\mathcal{A}_m$ is used by the schedule. Let us denote by $\pi(k, m)$ the index of the processor that processes task T_k in the allocation $\mathcal{A}_m$, and by $\Sigma(k, l, m)$ the set of oriented links used to send data from $\pi(k, m)$ to $\pi(l, m)$ in the allocation $\mathcal{A}_m$. Then, any solution of the GRAPH-THROUGHPUT problem satisfies the following set of constraints:

$$\begin{cases}
(1,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{k, \pi(k,m)=i} w_{i,k} \leqslant 1 \\
(2,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{P_i \to P_j} \sum_{\substack{(T_k,T_l)\in E_A, \\ (P_i,P_j)\in\Sigma(k,l,m)}} c_{i,j} \times data_{k,l} \leqslant 1 \\
(3,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{P_j \to P_i} \sum_{\substack{(T_k,T_l)\in E_A, \\ (P_j,P_i)\in\Sigma(k,l,m)}} c_{j,i} \times data_{k,l} \leqslant 1 \\
(4,m) \ \ \forall \mathcal{A}_m, & \alpha_m \geqslant 0
\end{cases}$$

Indeed, for any solution to the GRAPH-THROUGHPUT problem, the processing capability of each processor cannot be exceeded (constraint $(1,i)$); one-port constraints for sending $(2,i)$ and receiving $(3,i)$ messages must be fulfilled at any node. Conversely, from any solution of previous set of inequalities, one can derive a valid schedule, where $\sum \alpha_m$ messages are processed every time-unit (this is exactly Theorem 2). Thus, the solution of the following linear program provides an optimal solution of the GRAPH-THROUGHPUT problem:

$$\begin{array}{l}
\textsc{Maximize} \sum_m \alpha_m, \\
\textsc{under the constraints} \\
\begin{cases}
(1,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{k, \pi(k,m)=i} w_{i,k} \leqslant 1 \\
(2,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{P_i \to P_j} \sum_{\substack{(T_k,T_l)\in E_A, \\ (P_i,P_j)\in\Sigma(k,l,m)}} c_{i,j} \times data_{k,l} \leqslant 1 \\
(3,i) \ \ \forall P_i, & \displaystyle\sum_{\mathcal{A}_m} \alpha_m \sum_{P_j \to P_i} \sum_{\substack{(T_k,T_l)\in E_A, \\ (P_j,P_i)\in\Sigma(k,l,m)}} c_{j,i} \times data_{k,l} \leqslant 1 \\
(4,m) \ \ \forall \mathcal{A}_m, & \alpha_m \geqslant 0
\end{cases}
\end{array}$$

Let us denote by $\rho_{\max}$ the optimal value of the objective function. The previous linear program is of little practical interest since both the number of constraints and the number of variables are possibly exponential in the size of the original instance of the GRAPH-THROUGHPUT problem. Nevertheless, using linear programming theory [33], it is possible to prove that one of the optimal solution to the linear program is one instance of COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ).

Indeed, the linear program has $|\mathcal{A}| + 3|V_P|$ constraints, where $|\mathcal{A}|$ is the number of all allocations. There is a vertex V of the polyhedron defined by linear constraints which is optimal, and V is given by the solution of a $|\mathcal{A}| \times |\mathcal{A}|$ linear system, such that at vertex V, at least $|\mathcal{A}|$ inequalities among $|\mathcal{A}| + 3|V_P|$ are tight. Since only $3|V_P|$ constraints are not of the form $(4, m)$, we know that at least $|\mathcal{A}| - 3|V_P|$ constraints of the form $(4, m)$ are tight, i.e. that at most $3|V_P|$ allocations have a non-zero weight. Thus, there exists an optimal solution where at most $3|V_P|$ allocations are actually used.

In order to achieve the proof of the theorem, we need to bound the size of the weights of these allocations. Again, consider the optimal solution defined by vertex V, which is given by the solution of a $|\mathcal{A}| \times |\mathcal{A}|$ linear system, where at most $m \leqslant 3|V_P|$ constraints are not of the form $\alpha_i = 0$. Let us consider the $m \times m$ linear system containing non-trivial equations. The coefficients of both the matrix and the right-hand side are either 0, 1, or

$$\sum_{k, \pi(k,m)=i} w_{i,k} \leqslant |V_A| \max w_{i,k},$$

or

$$\sum_{P_i \to P_j} \sum_{\substack{(T_k, T_l) \in E_A, \\ (P_i, P_j) \in \Sigma(k,l,m)}} c_{i,j} \times data_{k,l} \leqslant |V_P||E_A| \max c_{i,j} \max data_{k,l}.$$

Let us set $\alpha_i = \frac{a_i}{b_i}$. Both a_i and b_i can be computed using Cramer's rule, and therefore, both a_i and b_i are the determinant of matrices A_i and B_i whose coefficients are bounded by

$$M = \max(1, |V_A| \max w_{i,k}, |V_P||E_A| \max c_{i,j} \max data_{k,l}).$$

Then,

$$\begin{array}{rcll} \det(A_i) & = & \prod_j \lambda_j & \text{where the } \lambda_j\text{'s are the eigenvalues of } A_i \\ & \leqslant & ||A_i||_2^m & \text{where } m \leqslant 3|V_P| \\ & \leqslant & (mM)^m & \text{see [17]} \\ & \leqslant & (3|V_P|M)^{3|V_P|} & \end{array}$$

Therefore

$$\log(a_i) \text{ and } \log(b_i) \leqslant 3|V_P|(\log 3 + \log(|V_P|) + \log(M)) \text{ and thus}$$

$$\log(a_i) + \log(b_i) \leqslant 6|V_P|(2 + \log(|V_P|) + \log(M)),$$

and the sizes of both a_i and b_i satisfy the constraints of the instance of COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ).

Thus, among the optimal solutions of the GRAPH-THROUGHPUT problem, there exists a solution which uses at most $3|V_P|$ allocations, whose weights $\alpha_i = \frac{a_i}{b_i}$ satisfy $\log a_i + \log b_i \leqslant 6|V_P|(2 + \log(|V_P|) + \log(M))$. Therefore, if there exists a solution to the GRAPH-THROUGHPUT problem with throughput $\rho_{\max}$, then there exists a solution to COMPACT-WEIGHTED-GRAPH-THROUGHPUT$(G_A, G_P, \rho_{\max})$.

■

We now show the intrinsic difficulty of the problem:

Theorem 6. *COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ) is NP-complete.*

The proof of this theorem is divided into two lemmas.

Lemma 4. *COMPACT-WEIGHTED-GRAPH-THROUGHPUT$(G_A, G_P, \rho) \in$ NP.*

Proof. In order to prove that COMPACT-WEIGHTED-GRAPH-THROUGHPUT$(G_A, G_P, \rho) \in$ NP, we use the set of allocations as a certificate. We know that the solution consists in at most $3|V_P|$ allocations, whose weights $\alpha_i = \frac{a_i}{b_i}$ satisfy $\log a_i + \log b_i \leqslant 6|V_P|(2 + \log(|V_P|) + \log(M))$, where
$M = \max(1, |V_A| \max w_{i,k}, |V_P||E_A| \max c_{i,j} \max data_{k,l})$. Theorem 2 asserts that it is possible to build a valid schedule of communications and task processing, that achieves the throughput $\sum_m \alpha_m$, using a weighted decomposition of the bipartite graph. The weights on the edges of the bipartite graph represent the overall

communication time between the P_i and P_j. In order to prove that COMPACT-WEIGHTED-GRAPH-THROUGHPUT$(G_A, G_P, \rho) \in$ NP, we only need to prove that both the bipartite graph and the processing times can be encoded in size polynomial to the size $\mathcal{S}$ of the original instance, i.e.

$$\mathcal{S} = |V_A| + |E_A| + |V_P| + |E_P| + \max \log(c_{i,j}) + \max \log(data_{k,l}) + \max \log(w_{i,k}).$$

The overall communication time $W(P_i, P_j)$ between processors P_i and P_j is given by

$$W(P_i, P_j) = \sum_m \alpha_m \sum_{(T_k,T_l)\in E_A,\ \ (P_i,P_j)\in\Sigma(k,l,m)} c_{i,j}\, data_{k,l}.$$

Thus,

$$\log(W(P_i, P_j)) \leqslant \log(3|V_P|) + 6|V_P|(\log(|V_P|) + \log(M)) + \log(E_A) + \max \log(c_{i,j}) + \max \log(data_{k,l}) = O(\mathcal{S}^2).$$

Similarly, the overall processing time $W(P_i)$ on processor P_i is given by

$$W(P_i) = \sum_m \alpha_m \sum_{\pi(k,m)=i} w_{i,k}.$$

Thus,

$$\log(W(P_i)) \leqslant \log(3|V_P|) + 6|V_P|(\log(|V_P|) + \log(M)) + \log(V_A|) + \max \log w_{i,k} = O(\mathcal{S}^2).$$

Therefore, since all the quantities used in Theorem 2 are polynomial in the size of the original instance of the problem, and we can build in polynomial time a valid schedule achieving throughput ρ. ∎

Lemma 5. *COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ) is complete.*

Proof. In order to prove that COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ) is complete, we use a reduction from MINIMUM-MULTIWAY-CUT, which is NP-complete (and even APX-complete) [1]. MINIMUM-MULTIWAY-CUT is the following decision problem:

Definition 23 (MINIMUM-MULTIWAY-CUT(G_M, S, t, B)). *Given a weighted platform graph $G_M = (V_M, E_M)$, a set $S \subset V_M$ of terminals, a weight function t on the edges and a rational bound B, is there a multiway cut, i.e. a set $E'_M \subset E_M$ such that the removal of E'_M from E_M disconnects each terminal from all the others, and*

$$\sum_{e\in E'_M} t(e) \leqslant B\ ?$$

In order to prove the completeness of COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ), we need to build from the original instance of MINIMUM-MULTIWAY-CUT(G_M, S, t, B) an instance of COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ) which has a solution if and only if the original instance of MINIMUM-MULTIWAY-CUT(G_M, S, t, B) has a solution. Consider the following instance of COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ):

- The application graph is built as follows. It has the same number of vertices and the same number of edges as G_M. Each (non-oriented) edge (V_k, V_l) (of weight $t(V_k, V_l)$) in G_M is transformed in an (oriented) edge $(T_{\min(k,l)}, T_{max(k,l)})$ (of weight $data_{k,l} = t(V_k, V_l)$) in G_A. The resulting graph is clearly oriented and acyclic, thus representing a valid application graph.

- The platform graph is built as follows (see Figure 18). It consists of $|S|+2$ processors $P_1, \ldots, P_{|S|}, P_a, P_b$. The capabilities of the edges of the platform graph depicted in Figure 18 are the following

$$\forall i, \quad c_{P_i,P_a} = 0, \quad c_{P_b,P_i} = 0 \text{ and } c_{P_a,P_b} = 1,$$

  all the other communication times being $+\infty$.

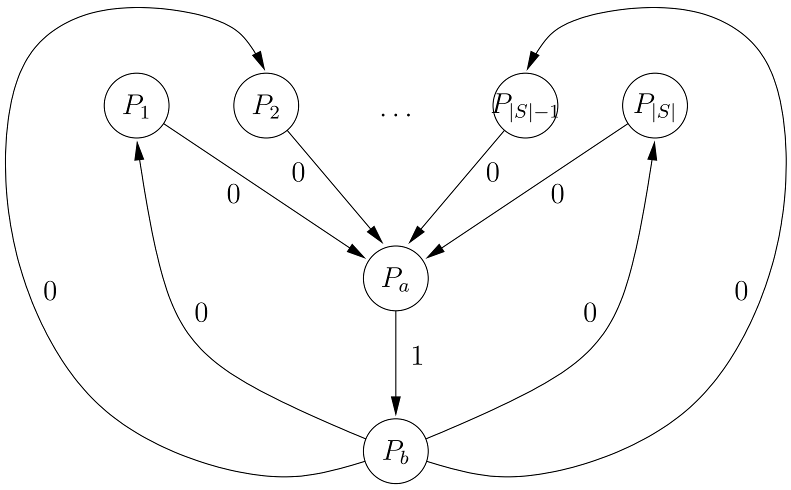


Figure 18: Platform graph for the reduction.

- The times to process the tasks of G_A on the processors of G_P are the following. If $V_k \in S$, then we will refer T_k as a "terminal task". Terminal task T_k is associated to terminal processor P_k, so that $w_{k,k} = 0$ and $w_{i,k} = +\infty$ if $i \neq k$. All the other tasks are not associated to a particular processor and can be processed in time 0 whatever the processor P_i executing it. Finally, P_a and P_b are unable to process any task ($w_{a,k} = w_{b,k} = +\infty$).

- We set $\rho = \frac{1}{B}$.

Let us first suppose that there is a solution to the original instance of MINIMUM-MULTIWAY-CUT(G_M, S, t, B), and let $\mathcal{C}_i$ denote the set of nodes connected to the terminal $V_i \in S$ in the graph $G_M = (V_M, E_M \setminus E'_M)$. Then, consider the following allocation (see Figure 19):

$$\forall T_k \in \mathcal{C}_i, T_k \text{ is processed on } P_i,$$

$$\forall (T_k, T_l) \in \mathcal{C}_i \times \mathcal{C}_j, \quad i \neq j, \qquad \Sigma(k,l) = \{(P_i, P_a), (P_a, P_b), (P_b, P_j)\}.$$

In this allocation, all processing costs are 0, since all terminal tasks are mapped on their terminal processor. All the incoming communication ports of the P_i's and that of P_b are 0. The same holds true for the outgoing communication port of the P_i's and of P_b. The outgoing port of P_a and the incoming port of P_b are busy during

$$\sum_{\substack{i \neq j,\\ (T_k,T_l) \in \mathcal{C}_i \times \mathcal{C}_j}} data_{k,l} \times c_{P_a,P_b} = \sum_{\substack{i \neq j,\\ (T_k,T_l) \in \mathcal{C}_i \times \mathcal{C}_j}} t_{k,l} = \sum_{(V_k,V_l) \in E'_M} t_{k,l} \leqslant B.$$

Thus, by Theorem 2, the platform G_P is able to process one application graph G_A every B time units using this allocation.

Suppose now that we have a solution to the instance of COMPACT-WEIGHTED-GRAPH-THROUGHPUT(G_A, G_P, ρ) that we have built, i.e. a collection of weighted allocations $(\mathcal{A}_1, \alpha_1), \ldots, (\mathcal{A}_m, \alpha_m)$ such that the platform G_P is able to process $\sum_m \alpha_m \geqslant \frac{1}{B}$ application graphs G_A every time unit. For every allocation, the fraction of time spent by any processor (P_i, P_a or P_b) is necessarily 0 since otherwise, the processing time would be infinite. The fraction of time spent by P_i or P_b sending data, and the fraction of time spent by P_i and P_a receiving data is 0 by construction.

The time spent by P_a sending tasks is equal to the time spent by P_b receiving tasks and is given by

$$\sum_m \alpha_m \sum_{\substack{(T_k,T_l),\\ (P_a,P_b) \in \Sigma(k,l,m)}} data_{k,l} \times c_{P_a,P_b} \leqslant 1.$$

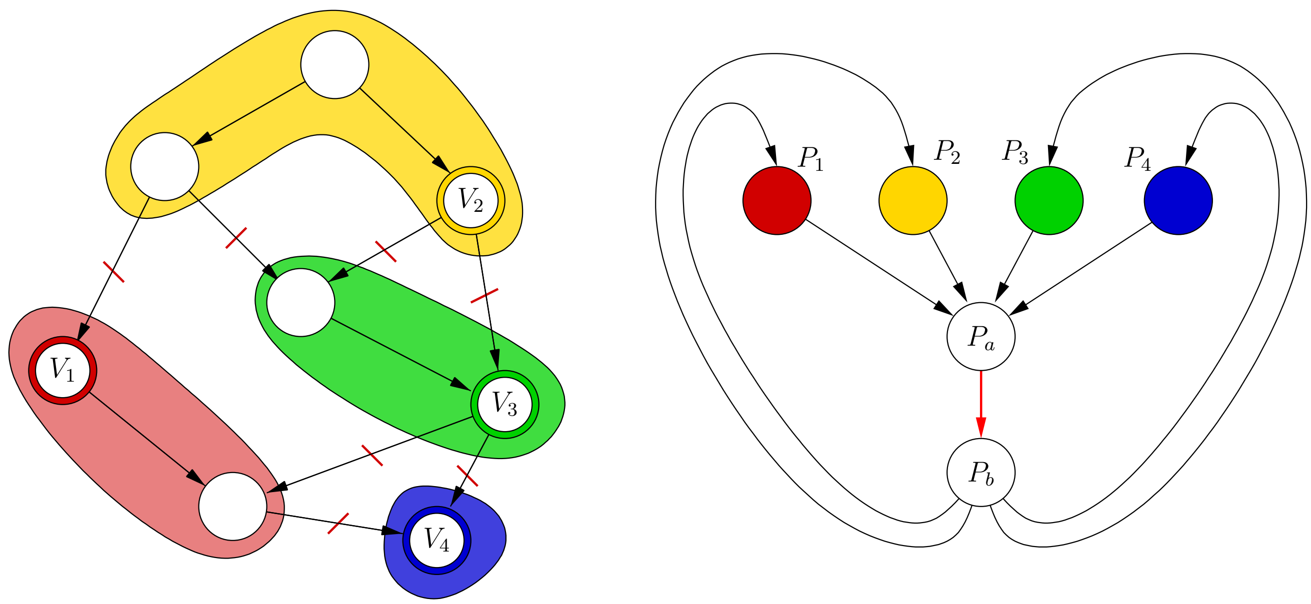


Figure 19: Reduction from the instance of MINIMUM-MULTIWAY-CUT: the weight of the cut is equal to the communication volume. On the left is the original (undirected) graph G_M which is also the (directed) application graph G_A. On the right the platform graph is represented. A processor P_i has the same color than the task T_i (corresponding to terminal V_i), which is the only task it can compute.

Consider the allocation $\mathcal{A}_m$, of weight α_m. Then

$$\sum_{\substack{(T_k,T_l),\\(P_a,P_b)\in\Sigma(k,l,m)}} data_{k,l}\times c_{P_a,P_b} = \sum_{\substack{(T_k,T_l),\\(P_a,P_b)\in\Sigma(k,l,m)}} t_{k,l}.$$

For every terminal processor P_i let $\mathcal{C}_i$ be the set of tasks in G_A processed on P_i. Clearly, $T_i \in \mathcal{C}_i$ (otherwise the overall processing time would be infinite).

Let us now set

$$E'_M = \Big\{(V_k,V_l)\in E_M, \quad (P_a,P_b)\in\Sigma(k,l,m)\Big\}.$$

Our aim is to prove that the removal of E'_M from E_M disconnects any terminal from all the others. Suppose by contradiction that there exists a (non-oriented) path between two terminals V_1 and V_2 in $(V_M, E_M \setminus E'_M)$ and consider the induced (oriented) path $T_1, T_{i_1}, \ldots, T_{i,k}, T_2$. T_{i_1} is either a predecessor or a successor of T_1 in the application graph G_A, so that if T_1 and T_{i_1} had not been mapped on the same processor, then either $(T_1, T_{i,1})$ or $(T_{i,1}, T_1)$ would have been removed. By a straightforward induction, we can therefore prove that there is no path between terminals in $(V_M, E_M \subset E'_M)$, and thus, every allocation induces a multiway cut in G_M.

Let us denote by a_m the overall weight of this multiway cut

$$a_m = \sum_{(T_k,T_l),\ (P_a,P_b)\in\Sigma(k,l,m)} t_{k,l},$$

and suppose (by contradiction) that

$$\forall m, \quad a_m > B.$$

Then,

$$\sum_m \alpha_m a_m > (\sum_m \alpha_m)B > 1,$$

what is absurd since

$$\sum_m \alpha_m a_m = \sum_m \alpha_m \sum_{\substack{(T_k,T_l),\\(P_a,P_b)\in\Sigma(k,l,m)}} data_{k,l} \times c_{P_a,P_b} \leqslant 1.$$

Thus, there exists m such that $a_m \leqslant B$: therefore, one of the allocation induces a multiway cut in G_M whose weight is less than B, thus providing a solution to MINIMUM-MULTIWAY-CUT(G_M, S, t, B). This achieves the proof of the NP-completeness of COMPACT-WEIGHTED-THROUGHPUT(G_A, G_P, ρ).

∎

6 Related problems

We classify several related papers along the following three main lines:

Scheduling task graphs on heterogeneous platforms Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors, see [31, 32, 43, 38, 12] among others. Unfortunately, all these heuristics assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications. Recent papers [22, 24, 40, 39] suggest to take communication contention into account. Among these extensions, scheduling heuristics under the one-port model [25, 26] are considered in [5]: just as in this paper, each processor can communicate with at most another processor at a given time-step.

Collective communications on heterogeneous platforms Several papers deal with the complexity of collective communications on heterogeneous platforms: broadcast and multicast operations are addressed in [11, 28], gather operations are studied in [20]. Broadcasting and multicasting on heterogeneous platforms have been studied under different models, in the context of heterogeneous target platforms.

Banikazemi et al. [2] consider a simple model in which the heterogeneity among processors is characterized by the speed of the sending processors. In this model, the interconnection network is fully connected (a complete graph). Some theoretical results (NP-completeness and approximation algorithms) have been proved for the problem of broadcasting a message under this model: see [19, 30, 29]. A more complex model is introduced in [3], where the time spent for the transfer through the network, and the time needed to receive the message are also taken into account.

Asymptotically optimal algorithms have been derived for series of broadcasts [8] and scatters [27] on an heterogeneous platform, under the communication model presented in Section 2. On the other hand, it has been proved in [7] that under the same communication model, optimizing the throughput of a series of multicasts in NP-Hard.

Master-slave on the computational grid Master-slave scheduling on the grid can be based on a network-flow approach [36, 35] or on an adaptive strategy [21]. Note that the network-flow approach of [36, 35] is possible only when using a full multiple-port model, where the number of simultaneous communications for a given node is not bounded. This approach has also been studied in [23]. Enabling frameworks to facilitate the implementation of master-slave tasking are described in [18, 44].

In [42], Taura and Chien prove that finding the best allocation, as defined in Section 2.5, but when restricting to a single allocation, i.e. when mapping all instances of a given task type onto the same processor, is NP Complete in the strong sense. In this paper, it is proven that the problem of finding an optimal linear combination of allocations can be solved in polynomial time for a large class of application graphs, but remains NP-Complete in the strong sense for general application graphs. We point out that linear combinations of allocations lead to better results, both from a practical and theoretical point of view. From a practical point of view, consider the case where the number of tasks in the application graph is much smaller than the number of processors in the task graph, which may hold true on large grid platforms. When using a single allocation and mapping each task on a single processor, several resources will be kept idle, thus leading to a poor throughput. From a theoretical point of view, as noted by Taura and Chien, the problem of finding the best allocation is NP-Complete even for simple application graphs. If we consider an application graph represented by a linear chain of tasks without any data files ($data_{k,l} = 0$ for every k, l), and m identical processors, then finding the best single allocation is equivalent to solving a bin packing problem with m bins. However, the dependency depth of this application graph is 1, and therefore, the algorithm proposed in Section 4 provides the best linear combination of allocations in polynomial time.

7 Conclusion

In this paper, we have dealt with the implementation of mixed task and data parallelism onto heterogeneous platforms. We have shown how to determine the best steady-state schedule in polynomial time for a large class of application graphs and for a arbitrary platform graphs, using a linear programming approach.

We have also derived several complexity results. We have shown that the problem of optimizing the steady-state throughput is NP-Complete in the general case. We have been able to formulate a compact version of the problem that belongs to the NP complexity class but which does not restrict the optimality of the solution.

This work can be extended in the following two directions:

- On the theoretical side, we could try to solve the problem of maximizing the number of tasks that can be executed within K time-steps, where K is a given time-bound. This scheduling problem is more complicated than the search for the best steady-state, but a smaller time period limits memory requirements and may be necessary in order to derive more dynamic schedules, where the allocations may change according to changes in platform capabilities.

- On the practical side, we need to run actual experiments rather than simulations. Indeed, it would be interesting to capture actual architecture and application parameters, and to compare heuristics on a real-life problem suite, such as those in [10, 41].

References

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

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

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

[4] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor grids. In *PARA'02: International Conference on Applied Parallel Computing*, LNCS 2367, pages 423–432. Springer Verlag, 2002.

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

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

[7] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Complexity results and heuristics for pipelined multicast operations on heterogeneous platforms. Research Report RR-2004-07, LIP, ENS Lyon, France, January 2004.

[8] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Pipelining broadcasts on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

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

[10] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002.

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

[12] Vincent Boudet and Yves Robert. Scheduling heuristics for heterogeneous processors. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 2109–2115. CSREA Press, 2001. Extended version available (on the Web) as Technical Report 2001-22, LIP, ENS Lyon.

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

[14] T. D. Braun, H. J. Siegel, and N. Beck. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel and Distributed Computing*, 61:810–837, 2001.

[15] L. Carter, H. Casanova, J. Ferrante, and B. Kreaseck. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2003*. IEEE Computer Society Press, 2003.

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

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

[18] J. P Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society Press, 2000.

[19] N.G. Hall, W.-P. Liu, and J.B. Sidney. Scheduling in broadcast networks. *Networks*, 32(14):233–253, 1998.

[20] J.-I. Hatta and S. Shibusawa. Scheduling algorithms for efficient gather operations in distributed heterogeneous systems. In *2000 International Conference on Parallel Processing (ICPP'2000)*. IEEE Computer Society Press, 2000.

[21] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

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

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

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

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

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

[27] A. Legrand, L. Marchal, and Y. Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. In *APDCM'2004, 6th Workshop on Advances in Parallel and Distributed Computational Models*. IEEE Computer Society Press, 2004.

[28] R. Libeskind-Hadas, J. R. K. Hartline, P. Boothe, G. Rae, and J. Swisher. On multicast algorithms for heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 61(11):1665–1679, 2001.

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

[30] P. Liu and T.-H. Sheng. Broadcast scheduling optimization for heterogeneous cluster systems. In *SPAA'2000, 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 129–136. ACM Press, 2000.

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

[32] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, LNCS 1123. Springer Verlag, 1996.

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

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

[35] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.

[36] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.

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

[38] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

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

[40] O. Sinnen and L. Sousa. Exploiting unused time-slots in list scheduling considering communication contention. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *EuroPar'2001 Parallel Processing*, pages 166–170. Springer-Verlag LNCS 2150, 2001.

[41] 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.

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

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

[44] J. B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.