Laboratoire de l'Informatique du Parallélisme

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

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Steady-state scheduling of task graphs on heterogeneous computing platforms

Olivier Beaumont,
Arnaud Legrand,
Loris Marchal,
Yves Robert

May 2003

Research Report N° 2003-29

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`

Steady-state scheduling of task graphs on heterogeneous computing platforms

Olivier Beaumont, Arnaud Legrand, Loris Marchal, Yves Robert

May 2003

Abstract

In this paper, we consider the execution of a complex application on a heterogeneous "grid" computing platform. The complex application consists of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks. There are dependences (precedence constraints) between these tasks. A typical example is the repeated execution of the same algorithm on several distinct data samples. We use a non-oriented graph to model the grid platform, where resources have different speeds of computation and communication. We show how to determine the optimal steady-state scheduling strategy for each processor (the fraction of time spent computing and the fraction of time spent communicating with each neighbor) and how to build such as schedule. This result holds for a quite general framework, allowing for cycles and multiple paths in the platform graph.

Keywords: Scheduling, steady state, task graphs, heterogeneous platforms

Résumé

Nous nous intéressons à l'ordonnancement d'une application complexe sur une plateforme de calcul hétérogène. L'application consiste en une suite de problèmes identiques et indépendants à résoudre. Chaque problème correspond à un graphe de tâches, avec contraintes de précédence. Un exemple typique de cette situation est l'exécution répétée d'un même algorithme sur des données différentes. La plateforme de calcul est modélisée par un graphe non-orienté, où les ressources ont des vitesses de calcul et de communication différentes. Nous montrons comment déterminer le régime permanent optimal pour chaque processeur (c'est-à-dire la fraction de temps passée à calculer et celles passées à communiquer avec chacun de ses voisins) et comment construire un tel ordonnancement.

Mots-clés: Ordonnancement, régime permanant, graphe de tâches, plateforme hétérogène

1 Introduction

In this paper, we consider the execution of a complex application, on a heterogeneous "grid" computing platform. The complex application consists of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks. 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 task graph depicted in Figure 1. This graph models the algorithm. We borrow this example from Subhlok et al. [24]. There is a main loop which is executed several times. Within each loop iteration, there are four tasks to be performed on some matrices. Each loop iteration is what we call a problem instance. Each problem instance operates on different data, but all instances share the same *task graph*, i.e. the acyclic graph of Figure 1. For each node in the task graph, there are as many task copies as there are iterations in the main loop.

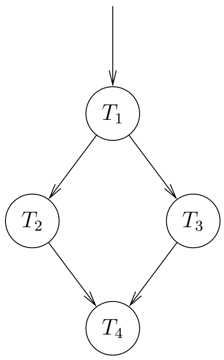


Figure 1: A simple task graph example.

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 2 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).

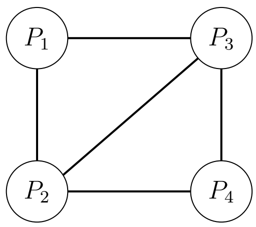


Figure 2: A simple platform example

Because the problems 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 eight 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 $i \neq j$).

Because the number of tasks to be executed on the computing platform is expected to be very large (otherwise why deploy the corresponding application on a distributed platform?), we focus on *steady-state* optimization problems rather than on standard *makespan* minimization problems. Minimizing the makespan, i.e. the total execution time, is a NP-hard problem in most practical situations [13, 22, 10], while it turns out that the optimal steady-state can be characterized very efficiently, with low-degree polynomial complexity. For our target application, the optimal steady state is defined as follows: for each processor, determine the fraction of time spent computing, and the fraction of time spent sending or receiving each type of tasks along each communication link, so that the (averaged) overall number of tasks processed at each time-step is maximum. In addition, we will prove that the optimal steady-state scheduling is very close to the absolute optimal scheduling: given a time bound K, it may be possible to execute more tasks than with the optimal steady-state scheduling, but only a constant (independent of K) number of such extra tasks. To summarize, steady-state scheduling is both easy to compute and implement, while asymptotically optimal, thereby providing a nice alternative to circumvent the difficulty of traditional scheduling.

Our application framework is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [19], factoring large numbers [11], the Mersenne prime search [16], and those distributed computing problems organized by companies such as Entropia [12]. Several papers [21, 20, 15, 14, 26, 4, 3] have recently revisited the master-slave paradigm for processor clusters or grids, but all these papers only deal with independent tasks. To the best of our knowledge, the algorithm presented in this paper is the first that allows precedence constraints in a heterogeneous framework. In other words, this paper represents a first step towards extending all the work on mixed task and data parallelism [24, 9, 17, 1, 25] towards heterogeneous platforms. The steady-state scheduling strategies considered in this paper could be directly useful to applicative frameworks such as DataCutter [5, 23].

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 to be solved. We start the study of the problem with a simplified version of the problem to present the main ideas: the task graph is reduced to one node. In Section 3, we provide the optimal solution to this problem when the task graph is a reduced to a simple node, using a linear programming approach. We give an algorithm to find a schedule that achieve this optimal throughput in Section 4. Then, we deal the same problem with any arbitrary task graph. In Section 5, we provide the optimal solution to this problem, using the same kind of method as in Section 3. We give an algorithm to find a schedule that achieve this optimal throughput in Section 6. Section 7 state the complexity of the previous method and give some insights for a practical implementation. Finally, we give some remarks and conclusions in Section 8.

2 Models

2.1 The application

- 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 $G^{(m)} = (V^{(m)}, E^{(m)})$ of the same *task graph* (V, E). The number $|V|$ of nodes in V is the number of task types. In the example of Figure 1, there are four task types, denoted as T_1, T_2, T_3 and T_4.

- Overall, there are $n.|V|$ 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 task graph. To this purpose, we introduce two fictitious tasks, T_{begin} which is connected to the roots of the task 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*.

- There are p nodes $P_1, P_2, \ldots, P_p$ that represent the processors. In the example of Figure 2 there are four processors, hence $p = 4$. See below for processor speeds and execution times.

- Each edge represents a physical interconnection. Each edge $e_{ij} : P_i \rightarrow P_j$ 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 can be dealt with, see [3, 2].

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 [7]. 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 $e_{k,end} : T_k \rightarrow T_{end}$. It is modeled by letting $data_{k,end} = +\infty$.

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 task 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 task 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.

3 Optimal steady-state for scheduling independent tasks on a general platform

In this section we focus on scheduling independent tasks. Such an application is modeled with a very simple graph like the one depicted on Figure 3(a). Nevertheless, everything written in this section and in Section 4 also hold true when the task graph is a tree.

3.1 Definitions

- For each edge $e_{k,l} : T_k \rightarrow T_l$ in the task graph and for each processor pair (P_i, P_j), we denote by $s(P_i \rightarrow P_j, e_{k,l})$ 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 $s(P_i \rightarrow P_j, e_{k,l})$ 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 $sent(P_i \rightarrow P_j, e_{k,l})$. We have the relation:

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

  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 $T_k \in V$ 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

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

3.2 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:

$$\forall P_i, \forall T_k \in V, 0 \leq \alpha(P_i, T_k) \leq 1 \tag{3}$$
$$\forall P_i, P_j, \forall e_{k,l} \in E, 0 \leq s(P_i \to P_j, e_{k,l}) \leq 1 \tag{4}$$

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

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

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:

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

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

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

3.3 Conservation laws

The last constraints deal with *conservation laws*: we state them formally, then we work out an example to help understand these constraints.

Consider a given processor P_i, and a given edge $e_{k,l}$ in the task 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:

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

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.

3.4 Computing the optimal steady-state

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:

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

Here is a summary of the linear program:

Steady-State Scheduling Problem SSSP(G)
Maximize

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

subject to

$$\left\{ \begin{array}{ll}
\forall i, \forall k, & 0 \leq \alpha(P_i, T_k) \leq 1 \\
\forall i, j, \forall e_{k,l} \in E, & 0 \leq s(P_i \to P_j, e_{k,l}) \leq 1 \\
\forall i, j, \forall e_{k,l} \in E, & s(P_i \to P_j, e_{k,l}) = sent(P_i \to P_j, e_{k,l}) \\
 & \qquad \times (data_{k,l} \times c_{i,j}) \\
\forall i, \forall k, & \alpha(P_i, T_k) = cons(P_i, T_k) \times w_{i,k} \\
\forall i, & \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_i \to P_j, e_{k,l}) \leq 1 \\
\forall i, & \sum_{P_j \in n(P_i)} \sum_{e_{k,l} \in E} s(P_j \to P_i, e_{k,l}) \leq 1 \\
\forall i, & \sum_{T_k \in V} \alpha(P_i, T_k) \leq 1 \\
\forall i, & cons(P_i, T_{begin}) = 0 \\
\forall i, j, \forall e_{k,end} & sent(P_i \to P_j, e_{k,end}) = 0 \\
\multicolumn{2}{l}{\forall i, \forall e_{k,l} \in E,} \\
\multicolumn{2}{l}{\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}) + cons(P_i, T_k) =} \\
\multicolumn{2}{l}{\quad \sum_{P_j \in n(P_i)} sent(P_i \to P_j, e_{k,l}) + cons(P_i, T_l)}
\end{array} \right.$$

We have the following result:

Lemma 1. $opt(G, K) \leq TP(G) \times K$

Proof. Consider an optimal scheduling. 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 task 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 equations hold true:

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

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

Going back to the original variables, we derive:

$$opt(G, K) = \sum_i t_{i,end}(K) \leq TP(G) \times K \qquad \blacksquare$$

Basically, Lemma 1 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 to come up with a well-defined scheduling algorithm based upon steady-state operation. Consider the following algorithm (assume K is large enough):

- Solve the linear program SSSP(G): compute the maximal throughput $TP(G)$, compute all the values $\alpha(P_i, T_k)$, $cons(P_i, T_k)$, $s(P_i \rightarrow P_j, e_{k,l})$ and $sent(P_i \rightarrow P_j, e_{k,l})$. Determine the time-period T. 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 task 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 $r = \lfloor \frac{K-I-J}{T} \rfloor$.

- Steady-state scheduling: during r periods of time T, operate the platform in steady-state, according to the solution of SSSP(G).
- 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).
- The number of tasks processed by this algorithm within K time-units is equal to $\mathit{steady}(G, K) = (r + 1) \times T \times TP(G)$.

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:

Theorem 1. *The previous scheduling algorithm based upon steady-state operation is asymptotically optimal:*

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

Proof. Using Lemma 1, $\mathit{opt}(G, K) \leq TP(G).K$. From the description of the algorithm, we have $\mathit{steady}(G, K) = ((r + 1)T).TP(G) \geq (K - I - J).TP(G)$, hence the result because I, J, T and $TP(G)$ are constants independent of K. ∎

Because we have a linear programming problem in rational numbers, we obtain rational values for all variables in polynomial time (polynomial in the sum of the sizes of the task graph and of the platform graph). When we have the optimal solution, we take the least common multiple of the denominators, and thus we derive an integer period for the steady-state operation. See Section 4 for the construction of a schedule that achieves those values.

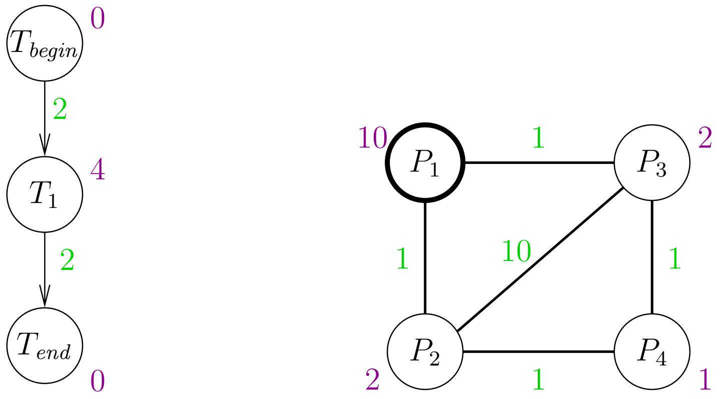
4 Reconstruction of an effective schedule for independent tasks on general platforms

Given the period and the number of tasks to be computed on each platform or transferred on each link, we need to exhibit a schedule that achieves those values. In this section, we give a polynomial algorithm to describe such a schedule for independent tasks and, we work out a simple example.

Let us consider the platform depicted on Figure 3(b) and a set independent tasks (whose characteristics are depicted on Figure 3(a)). We suppose that all the input files $e_{\mathit{begin},1}$ are on P_1 and that all output files $e_{1,\mathit{end}}$ 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 solution summarized on Figure 4 and build the *communication graph* depicted on Figure 4(b). Solid edges represent the transfers of $e_{\mathit{begin},1}$ and the dashed ones the transfers of $e_{1,\mathit{end}}$. 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.

We first transform the weighted directed graph describing the solution (Figure 4(b)) into a weighted bipartite graph (Figure 5(b)) by splitting each node into an incoming node (in white) and an outgoing node (in grey). We have assumed a *full overlap, single-port* operation mode,



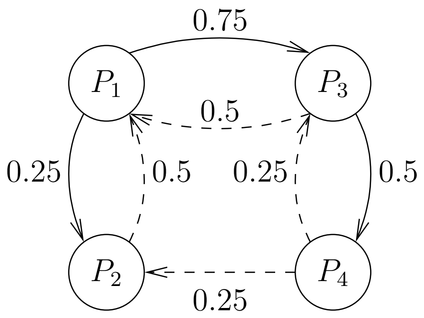
(a) Application graph

(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 3: A simple example for the matching algorithm

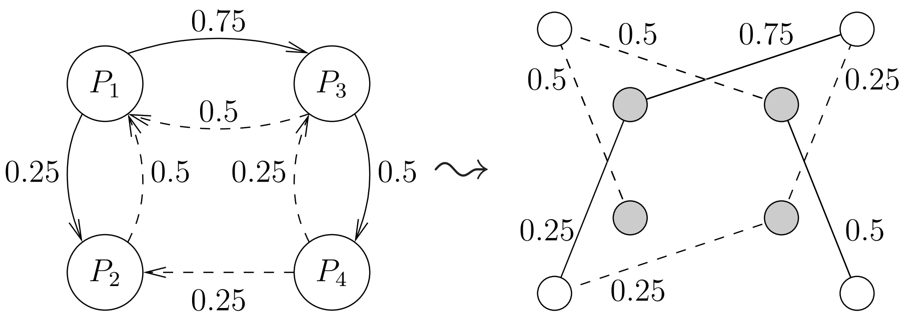
	$\alpha(P_i, T_1)$	$cons(P_i, T_1)$
P_1	100%	0.025
P_2	100%	0.125
P_3	100%	0.125
P_4	100%	0.250
Total	21 tasks every 40 seconds	

(a) Computation times



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

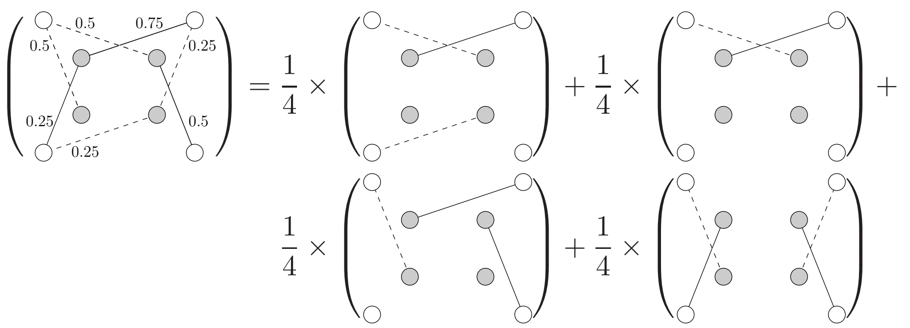
Figure 4: Solution of the linear program : the platform graph is annotated with the non-zero values of $s(P_i \to P_j, e_{k,l})$



(a) Solution of the linear program (b) Associated graph

Figure 5: Reconstruction of the solution

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. Thus, at a given time step, only communications corresponding to a matching in the bipartite graph can be done. Therefore, we need to decompose the weighted bipartite graph into a weighted sum of matchings such as the sum of the coefficients is smaller than one. For our example, we can use the following decomposition:



$$= \frac{1}{4}\chi_1 + \frac{1}{4}\chi_2 + \frac{1}{4}\chi_3 + \frac{1}{4}\chi_4 \tag{10}$$

In Equation (10), 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 in time $O(m^2)$, where m is the number of edges in the bipartite graph [18, vol.A chapter 20]. The number of matching is bounded by m. From these decomposition, it is then easy to build a schedule that achieves the optimal throughput (see Figure 6).

To summarize, the building of an effective schedule can be done by executing the following steps:

1. Solve the linear program

2. Transform the communication graph induced by the $s(P_i \to P_j, e_{k,l})$ into a weighted bipartite graph B

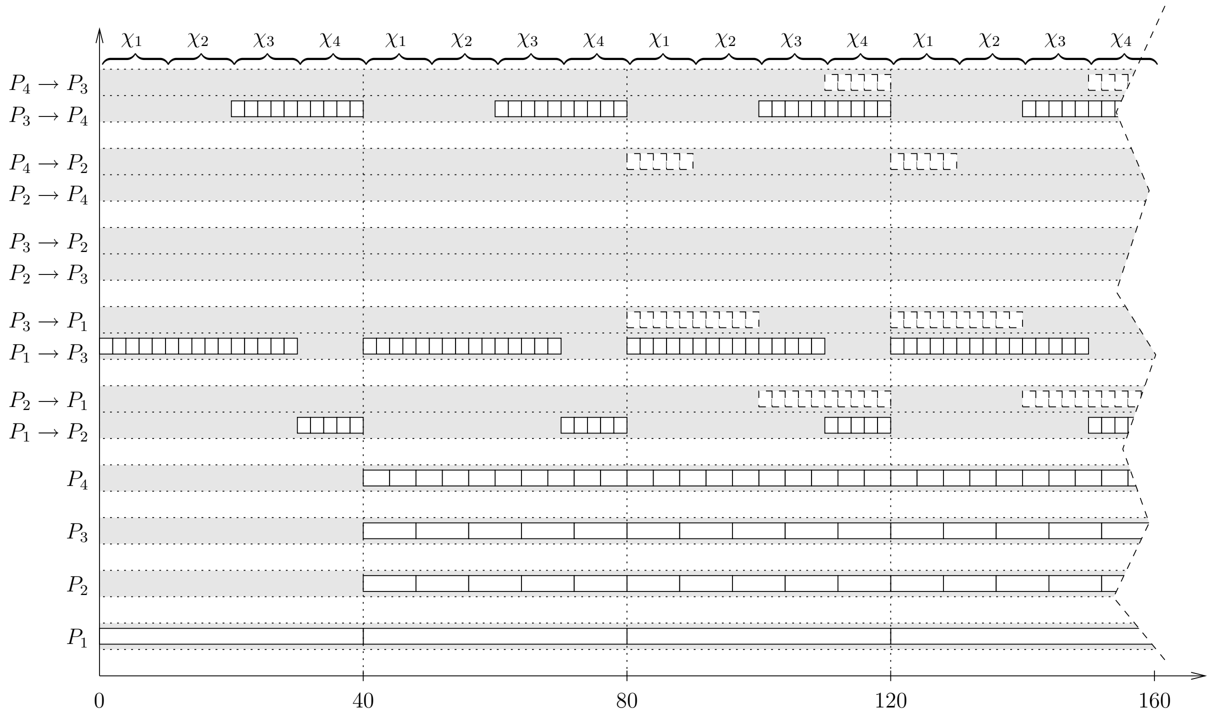


Figure 6: A schedule achieving the optimal throughput

3. Decompose the weighted bipartite graph B into a weighted sum of matchings $B = \sum \alpha_i \chi_i$ such as $\sum \alpha_i \leq 1$ (to ensure that all the communications can be done in time 1)

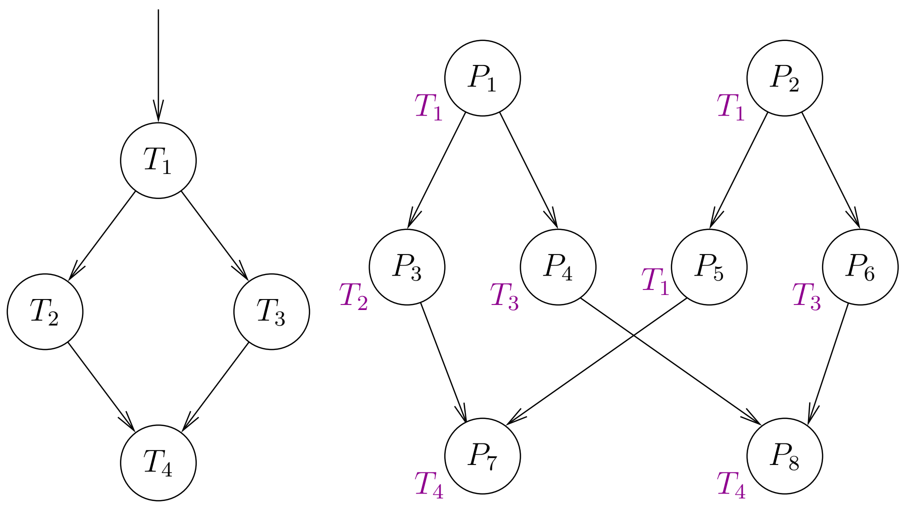
4. By construction of the matchings, each χ_i is a set of non conflicting communications of $e_{begin,1}$ or $e_{1,end}$ from a P_i to a P_j and then defines different events occurring during the schedule-period. At each of these events, the files present on P_i at the previous schedule-period are transferred. Note that there is only a polynomial number of events during the period, even though the length of the period is not polynomial in the problem size.

5 Optimal steady-state for scheduling a general DAG on a general platform

5.1 Why are DAGs more difficult to handle than trees or independent tasks

Scheduling a general DAG on a general platform turns out to be much more difficult than scheduling a tree or some independent tasks on a general platform.

For example, there is no feasible schedule of the application depicted on Figure 7(a) onto the platform depicted on Figure 7(b). Nevertheless, assuming that communication and communication times are all equal to one, if we had used the same equations as before, we would have get an expected throughput of 2 DAGs per time-unit. The difficulty arises from the join part of the graph. We need to merge data that corresponds to the same initial



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

Figure 7: Counter-example

instance of the task 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.

5.2 Adding constraints

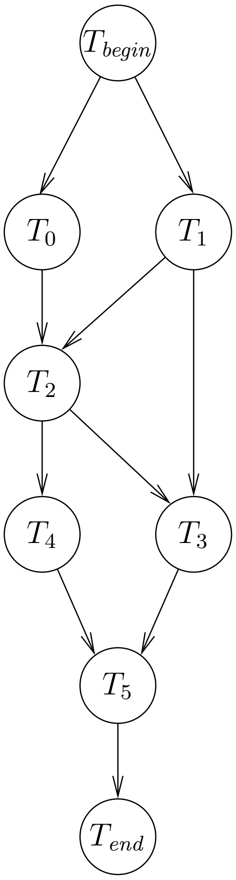
To avoid the problem exposed in the previous section, we keep track of the schedule by adding some informations to each variable. Therefore, $sent(P_i \to P_j, e_{k,l})$, $s(P_i \to P_j, e_{k,l})$, $\alpha(P_i, T_k)$ and $cons(P_i, T_k)$ will be annotated with a list of constraints L and be written $sent(P_i \to P_j, e^L_{k,l})$, $s(P_i \to P_j, e^L_{k,l})$, $\alpha(P_i, T^L_k)$ 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 list.

Definition 1. *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.*

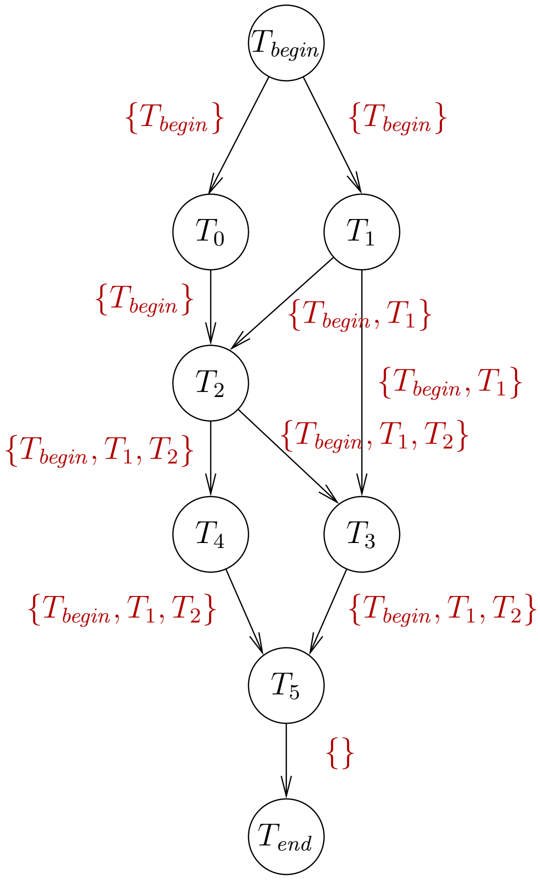
The *constraining tasks* of the $e_{k,l}$ are the tasks whose schedule is crucial to be memorized to ensure that join parts of the DAG will be done correctly. They can be constructed with Algorithm 1.

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 2. *A constraint list for an edge $e_{k,l}$ is a mapping from $\{T_{k_1}, \dots, T_{k_q}\}$ to $\{P_1, \dots, P_p\}$, where $\{T_{k_1}, \dots, T_{k_q}\}$ is set of constraining tasks of $e_{k,l}$. It is represented as a list of the form $\{T_{k_1} \to P_{i_1}, \dots, T_{k_q} \to P_{i_q}\}$*



(a) Application graph



(b) Annotating the application graph

Figure 8: Counter-example

```
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 1: Computing the constraining tasks.

Definition 3. *$Cnsts(e_{k,l}) = \{$ constraint list for $e_{k,l}$ $\}$*
$CnstsIn(T_k) = \{$ mapping from $\bigcup_{e_{l,k}}$(constraining tasks of $e_{l,k}$) to $\{P_1, \dots, P_p\}$ $\}$
$CnstsOut(T_k) = \begin{cases} Cnsts(e_{k,l}) & \textit{if there exists an } e_{k,l} \textit{ in } E \\ \emptyset & \textit{otherwise} \end{cases}$ *(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 $CnstsIn(T_k)$.

Definition 4. *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 two following definitions simply help to build the equations of the linear program.

Definition 5. *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 6. *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 $(T_k \rightarrow P_j) \in L$).*

5.3 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 $s(P_i \rightarrow P_j, e^L_{k,l})$ the (average) fraction of time spent each time-unit by P_i to send to P_j data involved by the edge $e_{k,l}$ under constraints L. As usual $s(P_i \rightarrow P_j, e^L_{k,l})$ is a nonnegative rational number. Let the (fractional) number of such files sent per time-unit be denoted as $sent(P_i \rightarrow P_j, e^L_{k,l})$. We have the same kind of relation as before:

$$s(P_i \rightarrow P_j, e^L_{k,l}) = sent(P_i \rightarrow P_j, e^L_{k,l}) \times (data_{k,l} \times c_{i,j}) \quad (11)$$

  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 $\alpha(P_i, T^L_k)$ the (average) fraction of time spent each time-unit by P_i to process tasks of type T_k fulfilling constraints L, and by $cons(P_i, T^L_k)$ the (fractional) number of tasks of type T_k fulfilling constraints L processed per time unit by processor P_i. We have the relation

$$\alpha(P_i, T^L_k) = cons(P_i, T^L_k) \times w_{i,k} \quad (12)$$

  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 $CnstsOut(T_k)$) of 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 $cons(P_i, T_k)$:

$$cons(P_i, T_k^L) = \sum_{\substack{L_2 \in \, CnstsIn(T_k) \\ T_k \text{ can be processed on } P_i \text{ under constraints } L_2 \\ L \text{ and } L_2 \text{ are compatible}}} prod(P_i, T_k^{L_2}) \qquad (13)$$

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:

$$\forall P_i, \forall T_k \in V, \forall L \in CnstsOut(T_k) \; 0 \leq \alpha(P_i, T_k^L) \leq 1 \qquad (14)$$

$$\forall P_i, P_j, \forall e_{k,l} \in E, \forall L \in Cnsts(e_{k,l}) \; 0 \leq s(P_i \to P_j, e_{k,l}^L) \leq 1 \qquad (15)$$

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

$$\forall P_i, \quad \sum_{\substack{P_j \in n(P_i) \\ e_{k,l} \in E \\ L \in Cnsts(e_{k,l})}} s(P_i \to P_j, e_{k,l}^L) \leq 1 \qquad (16)$$

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:

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

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

$$\forall P_i, \quad \sum_{\substack{T_k \in V \\ L \in CnstsOut(T_k)}} \alpha(P_i, T_k^L) \leq 1 \qquad (18)$$

5.4 Conservation laws

The last constraints deal with *conservation laws*. Consider a given processor P_i, and a given edge $e_{k,l}^L$ in the task 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

$\sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}^L)$ such files. Processor P_i itself executes some tasks T_k, namely $cons(P_i, T_k^L)$ 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:

$$\begin{aligned}
&\forall P_i, \forall e_{k,l} \in E : T_k \to T_l, \forall L \in Cnsts(e_{k,l}) \\
&\qquad \sum_{P_j \in n(P_i)} sent(P_j \to P_i, e_{k,l}^L) + cons(P_i, T_k^L) = \\
&\qquad\quad \sum_{P_j \in n(P_i)} sent(P_i \to P_j, e_{k,l}^L) + \sum_{\substack{L_2 \in CnstsIn(T_l) \\ L \text{ and } L_2 \text{ compatible}}} prod(P_i, T_l^{L_2})
\end{aligned} \tag{19}$$

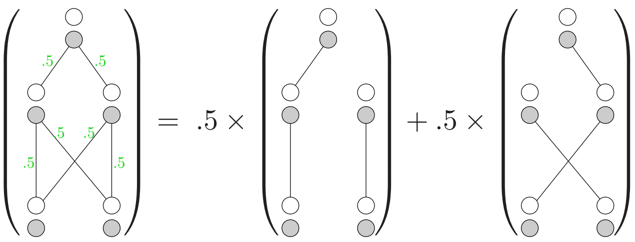
6 Reconstruction on an effective schedule for DAG on general platforms

Section 6.1 focuses on a simple example to explain why the schedule reconstruction is more difficult than in previous cases. Section 6.2 presents a way to decompose the solution of the linear program into a weighted sum of *scheduling schemes* and Section 6.3 shows how to mix those scheduling schemes to get an effective schedule.

6.1 Why are DAGs more difficult to handle than trees or independent tasks

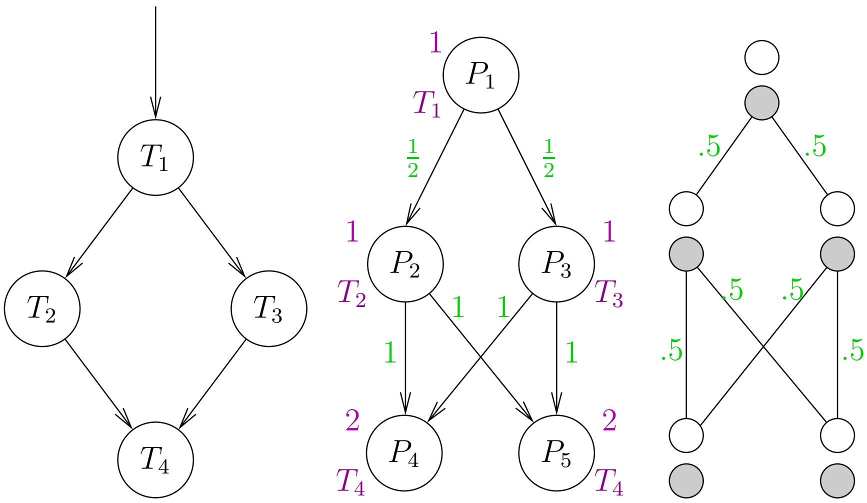
In the same way as computing the optimal throughput of a DAG on a general platform has turned out to be much more difficult than for simple tasks, reconstructing a schedule is a little bit more tricky, even with the constraints introduced in Section 5.2.

Using the same technique as in Section 4, we get the following decomposition.



(20)

Nevertheless, join parts have to be treated carefully. Figure 10 depicts two schedules. 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 task 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). Albeit, correct schedules can be built traversing the task graph backward (see Figure 10).



(a) Application graph. All the weights are identical

(b) Platform graph

(c) Bipartite graph transformation

Figure 9: Counter-example

6.2 Algorithm for decomposing the solution into scheduling schemes

The platform that we use to illustrate our algorithm in this section is depicted in Figure 3(b) and the application graph is depicted on Figure 11.

The rationale behind the interest in the throughput in steady-state is that mixing different schedules is likely to achieve an even better throughput that using a single one (which is furthermore difficult to find). This sections explains how to decompose the solution of the linear program into a weighted sum of scheduling schemes:

Definition 7. *A scheduling scheme is a copy of the task graph embedded in the platform graph such that each task is executed by a single processor. The weight of a scheduling scheme is the number of DAG copies that are executed per time-unit with the scheme.*

This is achieved by annotating the application graph with the non-zero values of $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$ and $sent(P_i \to P_j, e_{k,l}^L)$ (see Figure 12). The process is much easier when introducing $sent(P_i \to P_i, e_{k,l}^L)$, the amount of $e_{k,l}^L$ that are produced in place, that are not transferred to another processor and stay in place for another computation. Hence, we have:

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

$$sent(P_i \to P_i, e_{k,l}^L) = \sum_{\substack{L_2 \in CnstsIn(T_l) \\ T_l \text{ can be processed on } P_i \text{ under constraints } L_2 \\ L \text{ and } L_2 \text{ are compatible}}} prod(P_i, T_l^{L_2}) - \sum_{\substack{L_1 \in CnstsOut(T_k) \\ L \text{ and } L_1 \text{ are compatible}}} sent(P_j \to P_i, e_{k,l}^{L_1}) \quad (21)$$

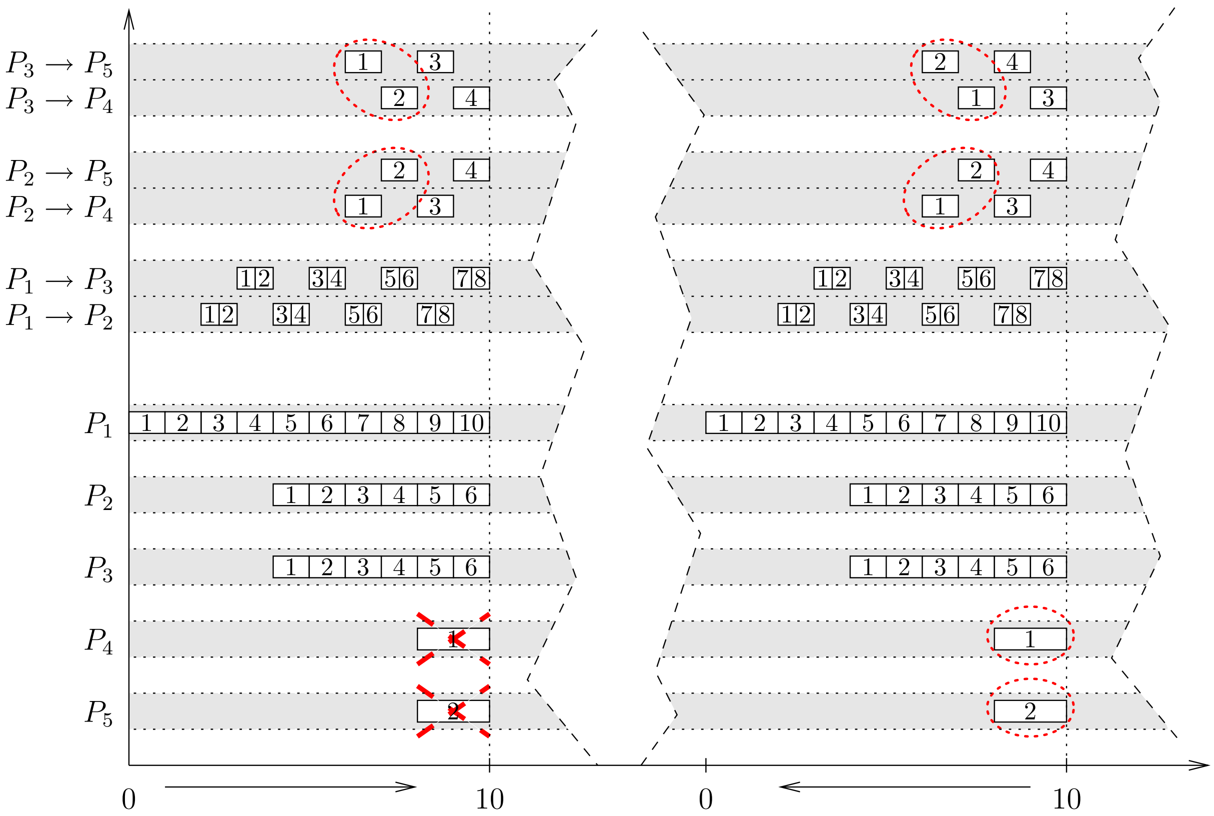


Figure 10: Effective schedules deduced from the decomposition of the bipartite graph

As explained in Section 6.1, we need to traverse the task graph backwards. Algorithm 2 builds a valid schedule from the solution of the linear program. The weight of this schedule, i.e. its throughput, is then equal to the minimum values over the $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$ and $sent(P_i \to P_j, e_{k,l}^L)$ involved in this schedule. The decomposition into a weighted sum of schedules then simply consists in finding a scheduling scheme, 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 schedules. Figure 13 depicts the main two (with the largest weight) schedules.

6.3 Using the matching algorithm to ensure that scheduling schemes are compatible

In this section, we explain how to mix the scheduling schemes obtained in the previous section to get an effective schedule. Going back to the example of Section 6.1, the optimal throughput is obtained by mixing two scheduling schemes depicted on Figure 14. To show that a real schedule can be computed from these scheduling schemes, we use the same approach as in Section 4.

A bipartite multigraph (see Figure 15) is associated to the platform graph depicted Figure 9(b) and to the two scheduling schemes depicted Figure 14. Each processor is split into an incoming node (in white) and an outgoing node (in grey). Each edge goes from a processor P_i to another processor P_j and is associated to an $e_{k,l}$ and a particular schedule. The weight of

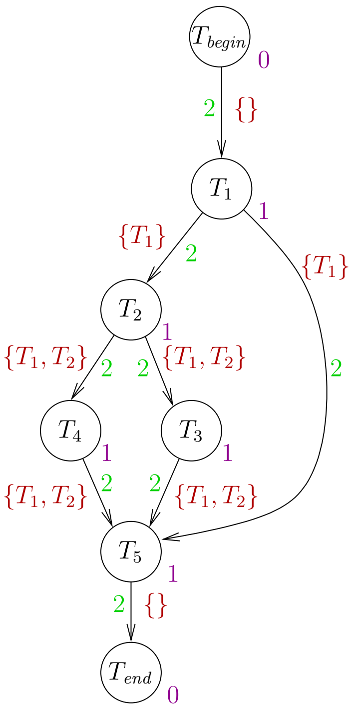
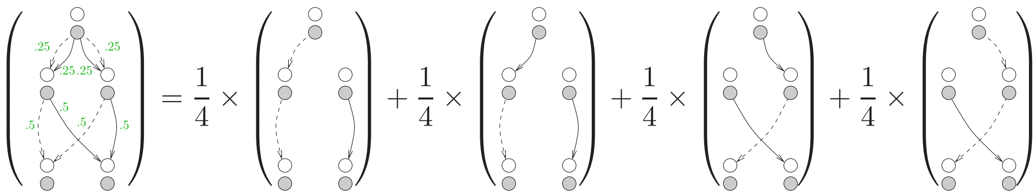


Figure 11: Not such a simple application graph

the edges is the fraction of time needed to transfer this file within this schedule from P_i to P_j. By construction, for a given node the sum of the weights of the adjacent edges is smaller than one. Using the algorithm described in [18, vol.A chapter 20], we can decompose this weighted bipartite multigraph into a weighted sum of matching such as the sum of the coefficients is smaller than one.



(22)

Therefore, we are able to compute a schedule achieving the optimal throughput, just like we did in Section 4.

7 Complexity and hints for a real implementation

7.1 Complexity

If we denote by n the number of tasks in the task graph and by p the number of nodes in the platform graph, the number of variables involved in the linear program of Section 5.3 may be proportional to $p^2 n^2 p^n$. Indeed, when dealing with $sent(P_i \rightarrow P_j, e_{k,l}^L)$, 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

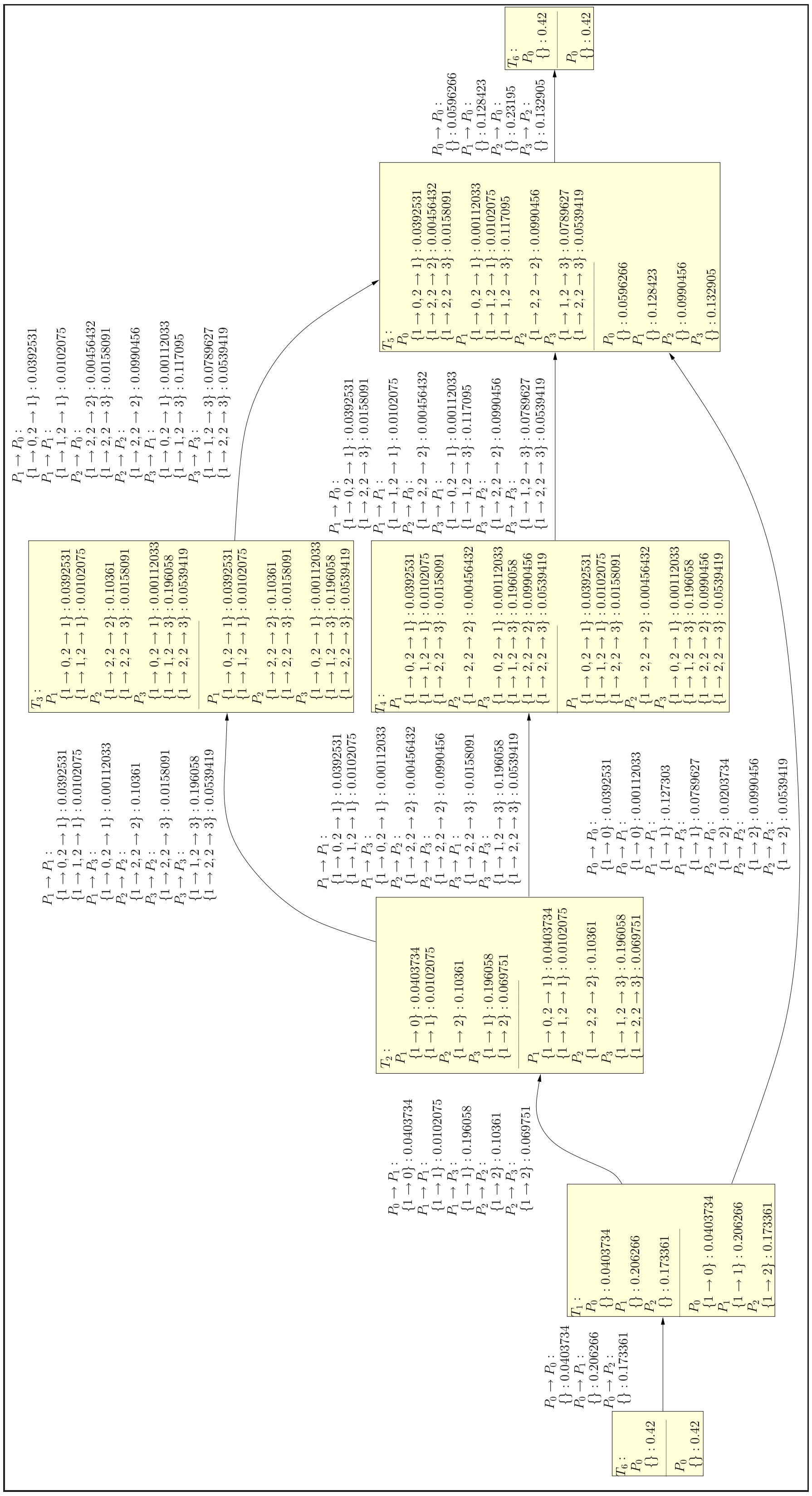
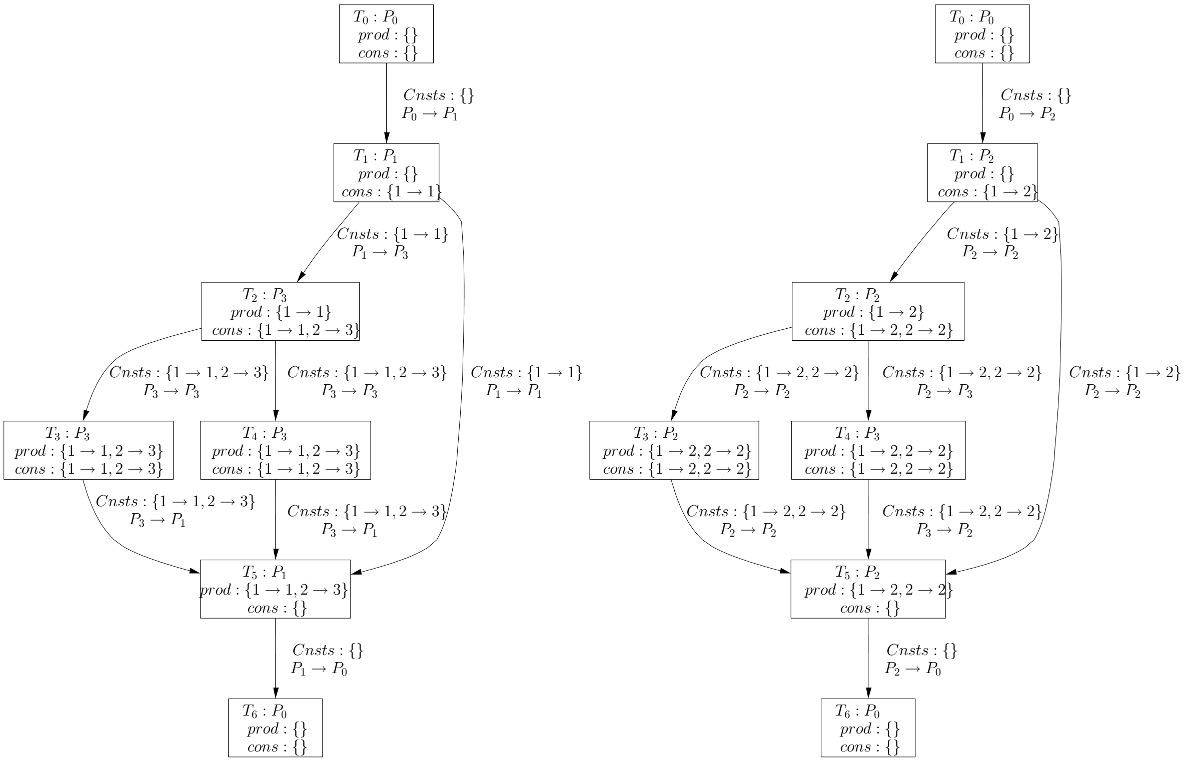


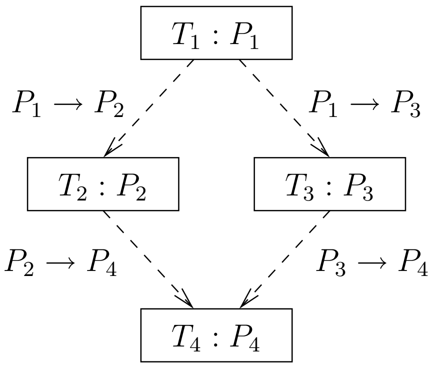
Figure 12: Solution given by the linear program : the application graph is annotated with the non-zero values of $cons(P_i, T_k^L)$, $prod(P_i, T_k^L)$, $sent(P_i \to P_j, e_{k,l}^L)$



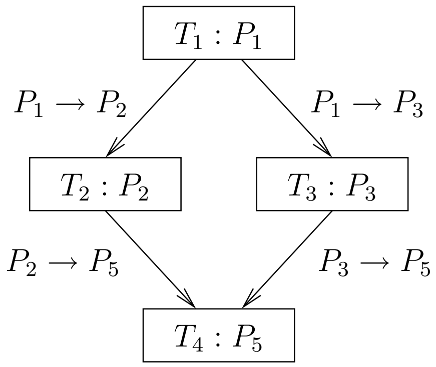
(a) Weight : 0.117095 (27.88%)

(b) Weight : 0.0990454 (23.58%)

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



(a) Weight : 0.5 (50.0%)



(b) Weight : 0.5 (50.0%)

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

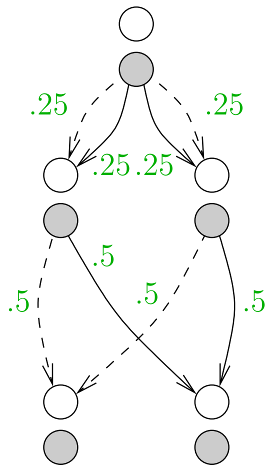


Figure 15: Bipartite graph associated to the platform graph depicted Figure 9(b) and to the scheduling schemes depicted Figure 14. Each edge goes from a processor P_i to another processor P_j and is associated to an $e_{k,l}$ and a particular schedule. The weight of the edges is the fraction of time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

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 2: Algorithm for building a scheduling scheme

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. Albeit, on Figure 17, there is at most one task in the constraint list.

Definition 8. *The dependency depth is the maximum number of constraining tasks of the* $e_{k,l}$.

Theorem 2. *Let $d \in \mathbb{N}$. For all application graph (G_a, V_a) whose dependency depth is bounded by d, and for all platform graph (G_p, V_p), it is possible to compute in polynomial time the optimal throughput and the optimal steady-state scheduling strategy that achieves this throughput.*

Proof. The algorithms described in Section 6 are polynomial and their inputs are spatially bounded by $p^2 n^2 p^d$. ■

7.2 Approximating the values

Let $S_1, \ldots, S_k$ be the schedules computed in Section 6.2 and $\alpha_1 = \frac{p_1}{T_p}, \ldots, \alpha_k = \frac{p_k}{T_p}$ be the throughput of these schedules. If we denote by α_{opt} the optimal steady-state throughput, we have

$$\alpha_{opt} = \sum_{i=1}^{q} \alpha_i \tag{23}$$

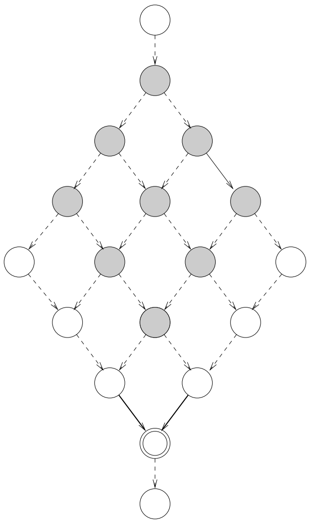


Figure 16: A 2D-mesh graph. All the gray tasks need to be memorized to ensure a correct reconstruction of the double-circled one.

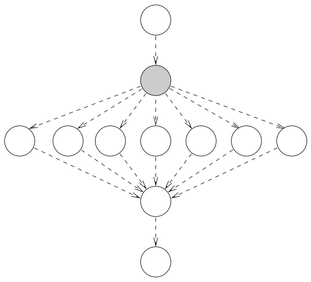


Figure 17: A fork graph. Only the gray task need to be memorized to ensure a correct reconstruction of the other tasks.

T_p may be very large and impracticable and rounding the weight of the schedules may be necessary. Let's 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 D.A.G.s that can be processed in steady-state using scheduling scheme S_i during a period T, we have :

$$r_i(T) = \lfloor \alpha_i T \rfloor \tag{24}$$

Note that, by using floor rounding, the equations of Section 5.3 still hold true and therefore lead to an effective schedule. We have the following equations:

$$\alpha_{opt} \geq \alpha(T) = \sum_{i=1}^{q} \frac{r_i(T)}{T} \geq \sum_{i=1}^{q} \frac{\alpha_i T - 1}{T} \geq \alpha_{opt} - \frac{q}{T} \tag{25}$$

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.*

7.3 Dynamic algorithm on general platforms

The algorithm presented in Section 6.3 to make the scheduling schemes compatible is not really usable in a real environment since it would require a global clock and a global synchronization. Nevertheless, a nice practical alternative is to use the 1D dynamic load balancing algorithm presented in [6] to decide which scheduling scheme should be used.

Let $S_1, \ldots, S_k$ be the schedules computed in Section 6.2 and $\alpha_1, \ldots, \alpha_k$ be the throughput of these schedules. We use the following dynamic programming algorithm:

```
DYNAMIC_ALLOCATION(α_1,...,α_p,B)
1: C = (c_1, ..., c_p) = (0, ..., 0)
2: for b=1..B do
3:     i ← argmin_{1≤j≤p}((c_j + 1)/α_j)
4:     A(b) ← i
5:     c_i ← c_i + 1
6: Return(A)
```

Algorithm 3: Dynamic programming algorithm for the optimal allocation of B independent identical chunks on p heterogeneous processors of relative speeds $\alpha_1, \dots, \alpha_p$.

For each value of $b \leq B$, let $C^{(b)} = (c_1^{(b)}, \dots, c_p^{(b)})$ denote the allocation of the first $b = \sum_{i=1}^{p} c_i$ chunks computed by the algorithm. This allocation is such as $\max \frac{c_i}{\alpha_i}$ is minimized [6]. Therefore, when allocating the scheduling schemes using the allocation A builded with this algorithm, we respect the proportion of the weights of the schedule 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 processors. Such an approach has already be used for scheduling independent tasks on tree-shaped platforms [8].

8 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 scheduling strategy for any task graph and for a general platform graph, using a linear programming approach.

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. Taking the initialization phase into account renders the problem quite challenging.

- 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 [5, 23].

References

[1] H. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[2] 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.

[3] 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. Extended version available as LIP Research Report 2001-25.

[4] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In D. S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, editors, *Cluster'2001*, pages 419–426. IEEE Computer Society Press, 2001. Extended version available as LIP Research Report 2001-13.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Chakrabarti, J. Demmel, and K. Yelick. Models and scheduling algorithms for mixed data and task parallel programs. *J. Parallel and Distributed Computing*, 47:168–184, 1997.

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

[11] James Cowie, Bruce Dodson, R.-Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Joerg Zayer. A world wide number field sieve factoring record: on to 512 bits. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - Asiacrypt '96*, volume 1163 of *LNCS*, pages 382–394. Springer Verlag, 1996.

[12] Entropia. URL: `http://www.entropia.com`.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[14] 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.

[15] 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.

[16] Prime. URL: `http://www.mersenne.org`.

[17] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

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

[19] SETI. URL: http://setiathome.ssl.berkeley.edu.

[20] 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.

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

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

[23] 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.

[24] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Fourth ACM SIGPLAN Symposium on Priciples & Practices of Parallel Programming*. ACM Press, May 1993.

[25] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel and Distributed Computing*, 60:297–319, 2000.

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