Scheduling multiple bags of tasks on heterogeneous master- worker platforms: centralized versus distributed solutions

Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, Yves Robert

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scheduling multiple bags of tasks on heterogeneous master-worker platforms: centralized versus distributed solutions

Olivier Beaumont — Larry Carter — Jeanne Ferrante — Arnaud Legrand — Loris Marchal — Yves Robert

N° 5739

Novembre 2005

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5739--FR+ENG

INRIA
RHÔNE-ALPES

Scheduling multiple bags of tasks on heterogeneous master-worker platforms: centralized versus distributed solutions

Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, Yves Robert

Thème NUM — Systèmes numériques
Projet Graal

Rapport de recherche n° 5739 — Novembre 2005 — 35 pages

Abstract: Multiple applications that execute concurrently on heterogeneous platforms compete for CPU and network resources. In this paper we consider the problem of scheduling applications to ensure fair and efficient execution on master-worker platforms where the communication is restricted to a tree embedded in the network. The goal of the scheduling is to obtain the best throughput while enforcing some fairness between applications.

We show how to derive an asymptotically optimal periodic schedule by solving a linear program expressing all problem constraints. For single-level trees, the optimal solution can be analytically computed. For large-scale platforms, gathering the global knowledge needed by the linear programming approach might be unrealistic. One solution is to adapt the multi-commodity flow algorithm of Awerbuch and Leighton, but it still requires some global knowledge. Thus, we also investigates heuristic solutions using only local information, and test them via simulations. The best of our heuristics achieves the optimal performance on about two-thirds of our test cases, but is far worse in a few cases.

Key-words: Parallel computing, scheduling, divisible load, multiple applications, resource sharing.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Ordonnancement d'applications concurrentes sur plate-forme maître-esclave hétérogène : comparaison des stratégies centralisées et distribuées

Résumé : Lorsqu'on exécute plusieurs applications simultanément sur une plate-forme de calcul hétérogène, celles-ci doivent se partager les ressources de calcul (processeurs) et de communication (bande-passante des liens réseau). Dans ce rapport nous nous intéressons à l'ordonnancement efficace et équitable de ces applications sur une plate-forme maître-esclave où les communications sont faites le long d'un arbre inclus dans le réseau.

Nous montrons qu'il est possible de calculer un ordonnancement périodique asymptotiquement optimal en utilisant la programmation linéaire. Pour les topologies en étoile (arbre de profondeur 1), nous montrons comment calculer la solution optimale de façon analytique. Pour des plates-formes de grande taille, rassembler l'information globale nécessaire au programme linéaire en un ordonnanceur centralisé peut sembler irréaliste. Une solution est d'adapter l'algorithme des flux concurrents d'Awerbuch et Leighton, mais celui nécessite tout de même quelques informations globales. Nous nous intéressons donc également aux heuristiques n'utilisant que des informations locales, et testons leurs performances par simulation. La meilleure de ces heuristiques atteint les performances optimales dans environ les deux tiers de nos essais, mais peut en être très éloigné dans quelques cas.

Mots-clés : Calcul distribué, ordonnancement, tâches divisibles,
applications multiples, partage de ressources.

1 Introduction

In this paper, we consider the problem of scheduling multiple applications that are executed concurrently, hence that compete for CPU and network resources, with fair management of those resources. The target computing platform is a master-worker architecture, either a simple one-level rooted tree platform, or a multi-level tree-shaped platform. In both cases we assume full heterogeneity of the resources, both for CPU speeds and link bandwidths.

Each application consists of a large collection of independent equal-sized tasks, and all originate at the unique master. The applications can be very different in nature, e.g. files to be processed, images to be analyzed or matrices to be manipulated. The resources required to execute tasks — both the communication volume and the computing demand — may well vary from one application to another. In fact, the relative *communication-to-computation ratio* of the applications proves to be an important parameter in the scheduling process.

This scenario is somewhat similar to that addressed by existing systems. For instance BOINC [19] is a centralized scheduler that distributes tasks for participating applications, such as SETI@home, ClimatePrediction.NET, and Einstein@Home. However, these applications all have a very low communication-to-computation ratio. For instance in Einstein@Home [23] a task is about 12 MB and requires between 5 and 24 hours of dedicated computation. Thus, the issue of network bandwidth is not important. However, our work anticipates that future applications, particularly ones that run on in-house computing networks, may have much higher communication requirements.

The scheduling problem is to maintain a balanced execution of all applications while using the computational and communication resources of the system effectively to maximize throughput. For each application, the master must decide which workers (i.e. which subtree) the tasks are sent to. For tree-shaped platforms, each non-leaf worker must make similar decisions: which tasks to compute in place, and which to forward to workers further down in the tree.

The scheduler must also ensure a fair management of the resources. If all tasks are equally important, the scheduler should try to process the same number of tasks for each application. We generalize this by allowing each application A_k to be assigned a *priority weight* $w^{(k)}$ that quantifies its relative worth. For instance, if $w^{(1)} = 3$ and $w^{(2)} = 1$, the scheduler should try to ensure that three tasks of A_1 are executed for each task of A_2.

For each application A_k, let $\nu^{(k)}(t)$ be the number of tasks of A_k completed by time t. At any given time t, we can define the throughput $\alpha^{(k)}$ to be $\nu^{(k)}(t)/t$. A natural objective would be to maximize the overall weighted throughput, namely $\sum_{k=1}^{K} \alpha^{(k)}/w^{(k)}$. However, such an objective could result in tasks of only one application being executed. To achieve a *fair* balance of execution, one should use the objective function:

$$\textsc{Maximize} \quad \min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\}. \tag{1}$$

This maximization of the weighted throughput, called *fair throughput* in the following, corresponds to the well-known MAX-MIN fairness strategy [13] between the different applications, with coefficients $1/w^{(k)}$, $1 \leqslant k \leqslant K$.

In this paper, we consider both centralized and decentralized schedulers. For smaller platforms it may be realistic to assume a centralized scheduler, which makes its decisions based upon complete and reliable knowledge of all application and platform parameters.

With such a knowledge at our disposal, we are able to determine the *optimal* schedule, i.e. the schedule which maximizes the weighted throughput of each application. The main idea is to gather all constraints into a linear program; the rational values output by the program will guide the construction of the actual periodic schedule. For single-level rooted trees, we provide an interesting characterization of the optimal solution: those applications with larger communication-to-computation ratio should be processed by the workers with larger bandwidths, independent of the communication-to-computation ratios of the workers.

For large-scale platforms, a centralized scheduler is unrealistic. Only local information is likely to be available to each participating resource. One major goal of this paper is to investigate whether decentralized scheduling algorithms can reach the optimal throughput, or at least achieve a significant fraction of it. In decentralized solutions, each resource makes decisions based upon limited local knowledge, typically the current capacity (CPU speed and link bandwidth) of its neighbors. From the theoretical perspective, one result of this paper is the adaption of the multi-commodity flow of Awerbuch and Leighton [2, 3] to provide a fully decentralized solution for scheduling multiple bags of tasks on tree-shaped platforms, assuming limited global knowledge of the solution size.

From a more practical point of view, we also provide several decentralized heuristics that rely exclusively on local information to make scheduling decisions. The key underlying principles of these heuristics is to give priority to fast communicating children, and to assign them applications of larger communication-to-computation ratios. A focus of this paper is evaluating decentralized heuristics through extensive simulations. The value of the optimal throughput (computed from the linear program) will serve as a reference basis to compare the heuristics.

The rest of the paper is organized as follows. In Section 2, we state precisely the scheduling problem under consideration, with all application and platform parameters, and the objective function. Section 3 explains how to analytically compute the best solution, using a linear programming approach, both for single-level trees (Section 3.1) and general tree-shaped (Section 3.2) platforms. In Section 3.3 we provide a decentralized solution to the scheduling problem using multi-commodity flows. Then Section 4 deals with the design of several decentralized scheduling heuristics, while Section 5 provides an experimental comparison of these heuristics, through extensive simulations. Section 6 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 7.

2 Platform and Application Model

2.1 Platform Model

The target platform is either a single-level tree (also called a *star network*) or an arbitrary tree. The master P_{master} is located at the root of the tree. There are p workers, $P_1, P_2, \ldots, P_p$; each worker P_u has a single parent $P_{p(u)}$, and the link between P_u and its parent has bandwidth b_u. We assume a linear-cost model, hence it takes X/b_u time-units to send a message of size X from $P_{p(u)}$ to P_u. The computational speed of worker P_u is c_u, meaning that P_u needs X/c_u time-units to execute X (floating-point) operations.

We assume that the master is not performing any computation, but it would be easy to modify this assumption. Indeed, we can add a fictitious extra worker paying no communication cost to simulate computation at the master.

There are several scenarios for the operation of the processors, which are discussed in Section 6. In this paper, we concentrate on the *full overlap, single-port model*. In this model, a processor node can simultaneously receive data from one of its neighbors, perform some (independent) computation, and send data to one of its neighbors. At any given time-step, there are at most two communications involving a given processor, one sent and the other received. More precisely: if $P_{p(u)}$ sends a message of size X to P_u at time-step t, then: (i) P_u cannot start executing or sending this task before time-step $t' = t + X/b_u$; (ii) P_u can not initiate another receive operation before time-step t' (but it can perform a send operation and independent computation); and (iii) $P_{p(u)}$ cannot initiate another send operation before time-step t' (but it can perform a receive operation and independent computation). Note that in the case of a star network, the master can only communicate to one worker at a time, while the workers can simultaneously receive data and perform independent computations.

2.2 Application Model

We consider K applications, A_k, $1 \leqslant k \leqslant K$. The master P_{master} initially holds all the input data necessary for each application A_k. Each application has a *priority weight* $w^{(k)}$ as described earlier.

Each application is composed of a set of independent, same-size tasks. We can think of each A_k as bag of tasks, and the tasks are files that require some processing. A task of application A_k is called a task of *type* k. We let $c^{(k)}$ be the amount of computation (in flops) required to process a task of type k. Similarly, $b^{(k)}$ is the size (in bytes) of (the file associated to) a task of type k. We assume that the only communication required is from the master to the workers, i.e. that the amount of data returned by the worker is negligible. Our results are equally applicable to the scenario in which the input to each task is negligible but the output is large. (We have not studied the case of large and different sizes for the input and output.) The *communication-to-computation ratio* of tasks of type k is defined as $b^{(k)}/c^{(k)}$ and represents the communication overhead per computation unit for application A_k.

Note our restriction that for each application, the amount of data and computation are identical for each task in the application. If an application has a finite set of different communication-to-computation ratios, and the distribution of these ratios are known, we can treat it as a set of separate applications and adjust the priority weights appropriately.

2.3 Objective Function

If we assumed that each application had an unlimited supply of tasks, our goal would be to maximize

$$\lim_{t\to\infty} \min_k \left\{ \frac{\nu^{(k)}(t)}{w^{(k)} \cdot t} \right\} \tag{2}$$

where $\nu^{(k)}(t)$ is the number of tasks of application A_k completed by time t.

However, we can do better than studying asymptotic behavior. Following standard practice, we partition time into a sequence of equal length time-steps, and optimize the "steady-state throughput". Thus, as stated earlier, our objective is:

$$\textsc{Maximize} \quad \min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\}. \tag{3}$$

where $\alpha^{(k)}$ is the number of tasks of A_k executed per time-step.

There are two features of this approach:

1. If we can derive an upper bound on the steady-state throughput for arbitrarily long time-steps, then this is an upper bound on the limit of formula (2).
2. If for a fixed length time-step we construct a *periodic schedule* – one that begins and ends in exactly the same state – and assuming it is possible to enter that state in a finite start-up period, then the periodic schedule's throughput will be a lower bound on the limit of formula (2).

As we shall see, this approach allows us to derive optimal results. The advantage over asymptotic analysis is we derive an upper bound for any length of time, and actually construct periodic schedules. When the number of tasks per application is large, the advantage of avoiding the NP-completeness of the makespan optimization problem outweighs the disadvantage of not knowing the exact length of the start-up and clean-up phases of a finite schedule. Many situations where this approach has been successful can be found in [10].

In reality, only integer numbers of tasks should be considered. However we relax the problem and deal with *rational* number of tasks in the steady-state equations that follow. The main motivation for this relaxation is technical: the scheduling is better amenable to an analytical solution if rounding problems are ignored. Thus, we consider that our applications are divisible load, i.e. they can be divided into smaller subtasks. See Section 6 for a comparison of our approach with *divisible load scheduling* [15, 16, 41]. From a practical point of view, this simplification is harmless: later in Section 3.1.2, we briefly explain how to retrieve an effective schedule with an integer number of tasks for each application.

3 Computing the Optimal Solution

In this section, we show how to compute the optimal throughput, using a linear programming formulation. For star networks we give a nice characterization of the solution, which will guide the design of some heuristics in Section 4. We also show how to state and solve the optimization for general tree-shaped platforms problem in terms of a multi-commodity flow approach.

3.1 Star networks

In this section we consider a star network. The master P_{master} has no processing capability and holds initial data for all tasks of any type. There are p worker processors P_u, $1 \leqslant u \leqslant p$. The platform model is the full-overlap single-port model.

3.1.1 Linear Program

Let $\alpha_u^{(k)}$ denote the rational number of tasks of type k executed by P_u every time-unit. Some of these quantities may well be zero, and if $\alpha_u^{(k)} = 0$ for all $1 \leqslant k \leqslant K$, then P_u is not enrolled in the computation. Each processor P_u computes an amount of $\sum_k \alpha_u^{(k)} \cdot c^{(k)}$ flops per time unit, hence the constraint $\sum_{k=1}^{K} \alpha_u^{(k)} \cdot c^{(k)} \leqslant c_u$.

The master processor sends $\sum_{k=1}^{K} \alpha_u^{(k)} \cdot b^{(k)}$ bytes to each worker P_u, which requires $\frac{\sum_k \alpha_u^{(k)} \cdot b^{(k)}}{b_u}$ time-units. These sends are sequentialized due to the one-port hypothesis, hence the constraint $\sum_{u=1}^{p} \frac{\sum_{k=1}^{K} \alpha_u^{(k)} \cdot b^{(k)}}{b_u} \leqslant 1$.

We have set $\alpha^{(k)} = \sum_{u=1}^{p} \alpha_u^{(k)}$, and we aim at maximizing the minimum of the weighted throughputs $\frac{\alpha^{(k)}}{w^{(k)}}$. Altogether, we have assembled the following linear program:

$$\begin{array}{l} \text{MAXIMIZE } \min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\}, \\ \text{UNDER THE CONSTRAINTS} \\ \left\{ \begin{array}{lll} \text{(4a)} & \forall k, & \displaystyle\sum_u \alpha_u^{(k)} = \alpha^{(k)} \\ \text{(4b)} & \forall u, & \displaystyle\sum_k \alpha_u^{(k)} \cdot c^{(k)} \leqslant c_u \\ \text{(4c)} & & \displaystyle\sum_u \frac{\sum_k \alpha_u^{(k)} \cdot b^{(k)}}{b_u} \leqslant 1 \\ \text{(4d)} & \forall k, u, & \alpha_u^{(k)} \geqslant 0 \end{array} \right. \end{array} \tag{4}$$

3.1.2 Reconstructing a Periodic Schedule

Suppose we have solved linear program (4). Note that conditions in the linear program deal with steady state behavior, so that it is not obvious that there exists a valid schedule, where precedence constraints are satisfied, that achieves obtained throughput. Nevertheless, we have determined all the values $\alpha_u^{(k)}$, which we write $\alpha_u^{(k)} = \frac{p_{u,k}}{q_{u,k}}$, where $p_{u,k}$ and $q_{u,k}$ are relatively prime integers. Let us define the period T_{period} as

$$T_{\text{period}} = \text{lcm}\{q_{u,k} | 1 \leqslant k \leqslant K, 1 \leqslant u \leqslant p\},$$

and $n_u^{(k)} = \alpha_u^{(k)} \cdot T_{\text{period}}$ for each worker P_u. If the master sends a bunch of $n_u^{(k)}$ tasks of type k to processor P_u for each k and each u every period, and if the processors greedily compute those tasks, we get a periodic schedule of period T_p whose throughput for each application type is exactly $\alpha^{(k)}$.

Of course the first and last periods will be different. We can assume an initialization phase, during which tasks are forwarded to processors, and no computation is performed. Then, during each time-period in steady state, each processor can simultaneously perform some computations, and send/receive some other tasks. Similarly, we need a clean-up phase in the end. But it can be shown that the previous periodic schedule is asymptotically optimal, among all possible schedules (not necessarily periodic). More precisely, given a time-bound B for the execution, it can be shown that the periodic schedule computes as many tasks of each type as the optimal, up to a constant (independent of B) number of tasks. This result is an easy generalization of the same result with a single application [6].

3.1.3 Structure of the Optimal Solution

We can prove that optimal solutions have a very particular structure:

Proposition 1.

- *Sort the link by bandwidth so that* $b_1 \geqslant b_2 \ldots \geqslant b_p$.
- *Sort the applications by communication-to-computation ratio so that* $\frac{b^{(1)}}{c^{(1)}} \geqslant \frac{b^{(2)}}{c^{(2)}} \ldots \geqslant \frac{b^{(K)}}{c^{(K)}}$.

Then there exist indices $a_0 \leqslant a_1 \ldots \leqslant a_K$, $a_0 = 1$, $a_{k-1} \leqslant a_k$ *for* $1 \leqslant k \leqslant K$, $a_K \leqslant p$, *such that only processors* P_u, $u \in [a_{k-1}, a_k]$, *execute tasks of type* k *in the optimal solution.*

In other words, each application is executed by a slice of consecutive workers. The most demanding application (in terms of communication-to-computation ratio) is executed by a first slice of processors, those with largest bandwidths. Then the next demanding application is executed by the next slice of processors. There is a possible overlap between the slices. For instance P_{a_1}, the processor at the boundary of the first two slices, may execute tasks for both applications A_1 and A_2 (and even for A_2 if $a_1 = a_2$).

Proof. Assume that there exists an optimal solution where there are two distinct processors P_u and P_v, where $u < v$, and two task types k and l, where $k < l$, such that

$$\alpha_u^{(k)} = 0, \alpha_u^{(l)} \neq 0, \alpha_v^{(k)} \neq 0, \alpha_v^{(l)} \neq 0$$

In other words, P_u executes tasks of type l but not k, but P_v execute both, which contradicts the proposition. Note that because $u < v$ and $k < l$:

$$b_u \geqslant b_v \text{ and } \frac{b^{(k)}}{c^{(k)}} \geqslant \frac{b^{(l)}}{c^{(l)}} \tag{5}$$

We build another solution, whose throughput is the same, but where either P_u will no longer execute tasks of type l, or P_v will no longer execute tasks of type k. The idea is that P_u and P_v exchange some tasks of type k and l. More precisely, letting $\widetilde{\alpha}$ instead of α denote the load units the new solution, we have:

$$P_u \begin{cases} \widetilde{\alpha}_u^{(k)} & \leftarrow \alpha_u^{(k)} + \beta_k = \beta_k \\ \widetilde{\alpha}_u^{(l)} & \leftarrow \alpha_u^{(l)} - \beta_l \end{cases} \quad \text{and} \quad P_v \begin{cases} \widetilde{\alpha}_v^{(k)} & \leftarrow \alpha_v^{(k)} - \beta_k \\ \widetilde{\alpha}_v^{(l)} & \leftarrow \alpha_v^{(l)} + \beta_l \end{cases}$$

The value of β_k and β_l is chosen as follows. The key idea is to impose

$$c^{(k)} \cdot \beta_k = c^{(l)} \cdot \beta_l,$$

so that the total computing time of P_u and P_v is left unchanged. Indeed for P_v, the time to compute tasks of type k and l is

$$\frac{c^{(k)} \cdot \widetilde{\alpha}_v^{(k)} + c^{(l)} \cdot \widetilde{\alpha}_v^{(l)}}{c_v} = \frac{c^{(k)} \cdot (\alpha_v^{(k)} - \beta_k) + c^{(l)} \cdot (\alpha_v^{(l)} + \beta_l)}{c_v} = \frac{c^{(k)} \cdot \alpha_v^{(k)} + c^{(l)} \cdot \alpha_v^{(l)}}{c_v}$$

The same holds for P_u. The constraints on β_k and β_l are

$$0 \leqslant \beta_k \leqslant \alpha_v^{(k)}, \quad 0 \leqslant \beta_l \leqslant \alpha_u^{(l)}$$

Therefore we let

$$\beta_l = \min\left(\alpha_u^{(l)}, \frac{c^{(k)}}{c^{(l)}} \cdot \alpha_v^{(k)}\right) \text{ and } \beta_k = \frac{c^{(l)}}{c^{(k)}} \cdot \beta_l$$

We claim that we still have a valid solution to the linear program (4). As already stated, the total computing time of P_u and P_v is left unchanged. Furthermore, the time spent by the master to communicate with P_u and P_v has not increased. Indeed, this time was equal to

$$M_{u,v} = \frac{\alpha_u^{(k)} \cdot b^{(k)} + \alpha_u^{(l)} \cdot b^{(l)}}{b_u} + \frac{\alpha_v^{(k)} \cdot b^{(k)} + \alpha_v^{(l)} \cdot b^{(l)}}{b_v}$$

The communication time is now equal to

$$\widetilde{M}_{u,v} = \frac{\widetilde{\alpha}_u^{(k)} \cdot b^{(k)} + \widetilde{\alpha}_u^{(l)} \cdot b^{(l)}}{b_u} + \frac{\widetilde{\alpha}_v^{(k)} \cdot b^{(k)} + \widetilde{\alpha}_v^{(l)} \cdot b^{(l)}}{b_v}$$

and we have

$$\widetilde{M}_{u,v} = M_{u,v} + \frac{\beta_k \cdot b^{(k)} - \beta_l \cdot b^{(l)}}{b_u} + \frac{\beta_l \cdot b^{(l)} - \beta_k \cdot b^{(k)}}{b_v}$$

hence

$$\widetilde{M}_{u,v} = M_{u,v} + \left(\beta_k \cdot b^{(k)} - \beta_l \cdot b^{(l)}\right)\left(\frac{1}{b_u} - \frac{1}{b_v}\right) = M_{u,v} + \beta_k \cdot c^{(k)}\left(\frac{b^{(k)}}{c^{(k)}} - \frac{b^{(l)}}{c^{(l)}}\right)\left(\frac{1}{b_u} - \frac{1}{b_v}\right)$$

Equation (5) shows that $\widetilde{M}_{u,v} \leqslant M_{u,v}$, which establishes the validity of the new solution.

In the case where $\beta_l = \alpha_u^{(l)}$, we have $\widetilde{\alpha}_u^{(l)} = 0$. Otherwise we have $\beta_k = \alpha_v^{(k)}$ and $\widetilde{\alpha}_v^{(k)} = 0$. In any case, this concludes the proof, by induction on the number of exchanges needed. □

Note that Proposition 1 would not hold if we had processor-task affinities, i.e. if the speed $c_u^{(k)}$ of a processor P_u would depend upon the task type k.

3.2 Tree-shaped Platforms

For tree-shaped platforms, we can also build a linear program that computes the optimal throughput. We use the following notations:

- P_{master} is the master processor
- $P_{p(u)}$ is the parent of node P_u for $u \neq$ master
- $\Gamma(u)$ is the set of indices of the children of node P_u
- $b_{u,v}$ is the bandwidth of the link $P_u \to P_v$ for $v \in \Gamma(u)$
- $sent_{u\to v}^{(k)}$ is the (fractional) number of tasks of type k sent by P_u to P_v, where $v \in \Gamma(u)$, every time-unit

We formulate the following linear program:

$$\begin{array}{l}
\textsc{Maximize } \min_k \left\{\frac{\alpha^{(k)}}{w^{(k)}}\right\}, \\
\textsc{under the constraints} \\
\left\{\begin{array}{lll}
\text{(6a)} & \forall k, & \displaystyle\sum_u \alpha_u^{(k)} = \alpha^{(k)} \\
\text{(6b)} & \forall k, \forall u \neq 0, & \displaystyle sent_{p(u)\to u}^{(k)} = \alpha_u^{(k)} + \sum_{v\in\Gamma(u)} sent_{u\to v}^{(k)} \\
\text{(6c)} & \forall u, & \displaystyle\sum_k \alpha_u^{(k)} \cdot c^{(k)} \leqslant c_u \\
\text{(6d)} & \forall u, & \displaystyle\sum_{v\in\Gamma(u)} \frac{\sum_k sent_{u\to v}^{(k)} \cdot b^{(k)}}{b_{u,v}} \leqslant 1 \\
\text{(6e)} & \forall k, u, & \alpha_u^{(k)} \geqslant 0 \\
\text{(6f)} & \forall k, u, v & sent_{u\to v}^{(k)} \geqslant 0
\end{array}\right.
\end{array} \tag{6}$$

The equations are similar to that of the program for star networks, except the second equation, which is new. This equation can be viewed as a conservation law. Every time-unit, P_u receives $sent^{(k)}_{p(u)\to u}$ tasks of type k. A fraction of these tasks, namely $\alpha^{(k)}_u$, are executed in place, while the remaining ones are forwarded to the children of P_u. It is important to understand that this conservation law really applies to the steady state operation; we do not have to detail which operation is performed at which time-step, because the tasks all commute, they are mutually independent. The reconstruction of an (asymptotically optimal) periodic schedule obeys exactly the same principles as in Section 3.1.2.

We did not succeed in deriving a counterpart of Proposition 1 for tree-shaped platforms. Intuitively, this is because the fast children of a node can themselves have slower children, so there is no *a priori* reason to delegate them the execution of the more demanding tasks. Still, we use the intuition provided by Proposition 1 to design the heuristic of Section 4.5.

3.3 Multi-commodity flows

In this section, we show how to compute the optimal solution for the general problem on tree-shaped platforms in a decentralized way. This approach is based on an algorithm by Awerbuch and Leighton [2, 3] for multi-commodity flows. The *multi-commodity flow problem* consists of shipping several different commodities from their source to their destination through a network such that the total flow going through any edge does not exceed its capacity. A *demand* is associated to each commodity. The objective if to maximize a fraction z such that z percent of each commodity is satisfied and the capacity constraints are not violated.

The Awerbuch and Leighton algorithm provides a distributed solution to the problem. Our objective is to adapt their algorithm to our scheduling problem while keeping the convergence result. The general idea is to map each type of application into a commodity to be shipped on the network directly made from the platform graph. Several difficulties have to be be circumvented:

1. First, the *multi-commodity flow problem* takes only link constraints into account: there is no notion of computation. We have to simulate the computing limitation of each node by adding fictitious links from computing nodes to (fictitious) computing sinks. The capacity of these new links is the computing power of the corresponding node.

2. In the *multi-commodity flow problem*, the constraints on the edges are simple: the sum of every flow going through one edge is less than its global capacity. In our adaptation, mainly due to the fictitious edges added to represent computing power, we have to consider more complex constraints like: the sum of each flow times a coefficient (depending on the flow and the edge) is less than the capacity of the edge. This leads to two problems:

   - we have to adapt the local optimization algorithm which gives the values of each commodity to ship on a given edge for a given time-step;
   - with this new optimization method, we have to prove that the convergence of the algorithm is still ensured.

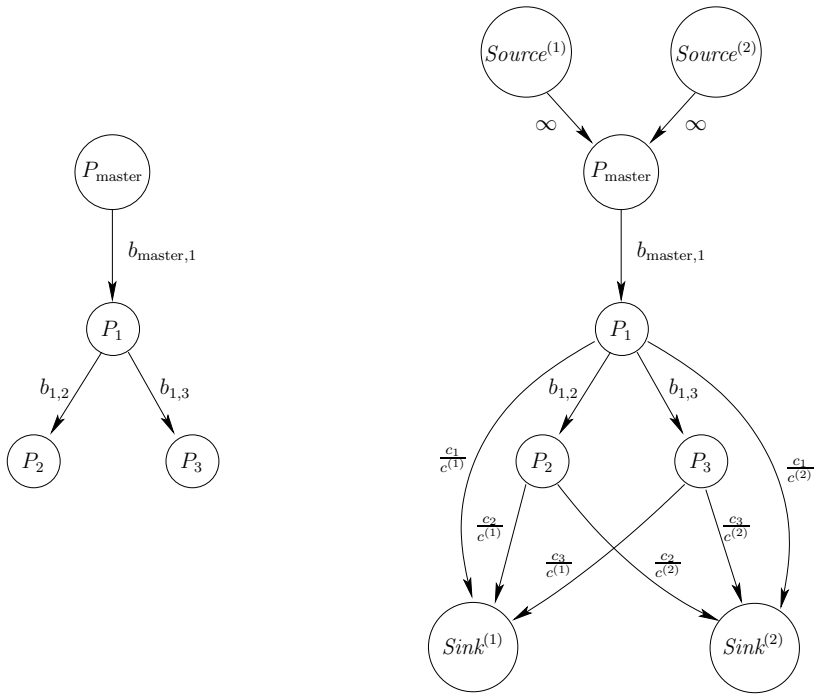
3. Last, the classical *multi-commodity flow problem* works in the multi-port model: the flow is limited only through each edge, not through the sending/receiving ports of each node. We have to enforce the one-port constraints into the multi-port model.

3.3.1 Problem formulation

In order to take computation power into account, we modify the model presented in Section 3.2 by adding some nodes and edges to the platform graph. We first add a sink node $Sink^{(k)}$ for each application of type k. Each processor P_u is linked to the sink $Sink^{(k)}$ by a link with capacity $\frac{c_u}{c^{(k)}}$ (the inverse of the time needed by P_u to process a task of type k).

To stick to the the model used in [3], we also add one source node $Source^{(k)}$ for each application type k. The source $Source^{(k)}$ for tasks of type k is linked to the master processor with an infinite capacity link.

These new nodes are "fictitious" in the sense that they do not represent any real node of the physical platform. An example of this adaptation is represented on Figure 1.



(a) original platform graph

(b) its transformation to apply multi-commodity flow techniques

Figure 1: An example of fictitious sources and sinks

In order to keep notations homogeneous, we denote by

- $sent^{(k)}_{p(u)\to u}$ the number of tasks of type k between the parent node $P_{p(u)}$ of P_u
- $sent^{(k)}_{Source^{(k)}\to P_0}$ the number of tasks of type k injected in the system
- $sent^{(k)}_{u\to Sink^{(k)}}$ the number of tasks of type k processed by P_u.

With those notations, we can check that the following equations hold true:

$$\begin{cases} \forall k, u \neq P_0, Source, Sink & \sum_{v\in\Gamma(u)} sent^{(k)}_{u\to v} + sent^{(k)}_{u\to sink(k)} = sent^{(k)}_{p(u)\to u} \\ & \text{tasks of type } k \text{ are either transmitted or processed locally at } P_u \\ \forall k, \quad sent^{(k)}_{Source^{(k)}\to P_0} = & \sum_u sent^{(k)}_{u\to Sink^{(k)}} \\ & \text{all injected tasks are processed by some } P_u \end{cases}$$

Therefore, $\forall k$, the values $\left(sent^{(k)}_{*\to *}\right)$ define a flow of value $sent^{(k)}_{Source^{(k)}\to P_0}$ between $Source^{(k)}$ and $Sink^{(k)}$.

We still have to express constraints for the computational limit of each processor, and for the one-port model. The amount of tasks of type k processed by a processor P_u is $sent^{(k)}_{u\to Sink^{(k)}}$, the constraint for computation on node P_u can be expressed as:

$$\sum_k sent^{(k)}_{u\to Sink^{(k)}} \times c^{(k)} \leqslant c_u$$

As we target a tree-shaped platform, the one-port model has to be enforced only for outgoing communications. On node P_u, the bound on outgoing communications is:

$$\sum_k \sum_{v\in\Gamma(u)} \frac{sent^{(k)}_{u\to v} \cdot b^{(k)}}{b_{u,v}} \leqslant 1$$

So on a node P_u, all constraints can be expressed as a weighted sum of some outgoing commodities, with general form:

$$\sum_{(v,k)\in\mathcal{S}} sent^{(k)}_{u\to v} \times \gamma_{u,v,k} \leqslant 1$$

where $\mathcal{S}$ is a given set of indices and $\gamma_{u,v,k}$ is some given coefficient (which depends upon the type of the constraint). This formulation is slightly more complex than the one for classical flows used in the *multi-commodity flow problem*:

$$\text{on each edge } (u,v), \quad \sum_k sent^{(k)}_{u\to v} \leqslant 1$$

Nevertheless, we will prove in next sections that it is possible to adapt the Awerbuch-Leighton algorithm for multi-commodity flow problems to our problem. In the following, we first present the Awerbuch-Leighton algorithm in Section 3.3.2, and then show how to adapt it in Section 3.3.3 to take the new constraints into account.

3.3.2 The Awerbuch-Leighton Algorithm

Main Result Consider a multi-commodity flow problem (with classical capacity constraints on the edges) on a directed graph $G = (V, E)$. Let us assume that there exists, for each commodity k, at any time, a way to ship $(1 + 2\varepsilon)d_k$ units of flow from the source node for commodity k to the sink node for commodity k.

In [3], Awerbuch and Leighton present an algorithm which is able to ship $(1+\varepsilon)d_k$ units of flow for each commodity k. Note that the assumption states that there is at any time a way to

ship the flow (in fact, even slightly more flow than what is shipped using Awerbuch Leighton algorithm), but the paths used to ship the flow may well change during the whole process. Surprisingly enough, the algorithm is very simple, fully distributed, with local control, and is able to cope with changes in link and node capacities, provided that above mentioned assumption holds true during the whole process.

Moreover, as we will prove in Section 3.3.3, the framework presented for multi-commodity flow problems in [3] can be generalized to a more general class of problems, such as the one considered in this paper.

Sketch of the algorithm Let us assume that for each source, there exists a single outgoing edge. This can be done, as noted in Section 3.3.1 by adding fictitious nodes linked with infinite capacity edges.

Each node is given some buffers to store messages, and there is one such buffer at each node associated to each directed edge and to each commodity. So a node with x incoming or outgoing edges will have $x \times K$ buffers. In addition to these *regular* buffers, the source node for commodity k is also given an *overflow* buffer for commodity k. The size of the regular buffer is bounded at this node: it cannot contain more than Q_k elements. If the source nodes has to store more than Q_k elements, the remaining elements are stored into the overflow buffer. We set (the exact value for Q_k can be found in [40])

$$Q_k = O\left(\frac{nd_k \ln(\frac{K}{\varepsilon})}{\varepsilon}\right) \tag{7}$$

The main idea of the algorithm is to introduce a potential function associated to each regular and overflow buffer. More precisely, the potential of a regular buffer with q elements for commodity k is given by

$$\phi_k(q) = \exp(\gamma_k q), \text{ where } \gamma_k = \frac{\varepsilon}{8nd_k},$$

and the potential of the overflow buffer for commodity k at node $Source^{(k)}$ is given by

$$\psi_k(q) = q\gamma_k \exp(\gamma_k Q_k)$$

The sketch of the algorithm is as follows. The time is divided into rounds and each round consists in the following four phases.

Phase 1: For each source $Source^{(k)}$, add $(1+\varepsilon)d_k$ units of flow to the overflow buffer of commodity k. Then, move as much flow as possible from the overflow buffer to the regular buffer at source node $Source^{(k)}$.

Phase 2: For each edge, push flow along the edge so as to minimize the sum of the potential of all the buffers associated to this edge. This requires to solve a multi-variate non-linear optimization problem. We will not detail at this step the resolution of this optimization problem, but we will present in Section 3.3.3 a general framework for solving a larger class of problems.

Phase 3: For each commodity k, empty the regular buffer for commodity k at the sink node $Sink^{(k)}$ for commodity k.

Phase 4: At each node P_u and for each commodity k, balance the buffers for commodity k associated to all incoming and outgoing edges, such that the sizes of all the buffers for commodity k are the same at node P_u.

Sketch of the proof The algorithm presented above does not guarantee that all units of flow will be shipped to their destination at the end of the algorithm. Nevertheless, it guarantees that the overall number of tasks that do not reach their destination remains bounded at any time. Since $(1+\varepsilon)d_k$ units of flow are injected in the system at each round, this means that the throughput for commodity k obtained after a large number of rounds is arbitrarily close to d_k.

The proof is based on a precise analysis of the overall potential of the system, i.e. the sum of the potentials associated to all regular and overflow buffers in the system. During the different phases in one round, the evolution of the overall potential is the following:

- Phase 1 increases the value of the overall potential since flow units are added in regular (and possibly overflow) buffers at each source, and potential functions are increasing functions in the size of the buffers.
- Phase 2 decreases the value of the overall potential since it aims at minimizing independently the potential of the buffers at the tail and head of each edge, for each commodity.
- Phase 3 decreases the value of the overall potential since some flow units are removed from the buffers (at sink nodes), and potential functions are increasing functions in the size of the buffers.
- At last, phase 4 decreases the overall potential since for each commodity, it aims at balancing the buffers at each node, and the potential is a convex function in the size of the buffers.

More precisely, it can be proved (see [3, 40]) that the overall potential increase for commodity k during phase 1 is bounded by

$$(1+\varepsilon)d_k\phi'(s_k),$$

where s_k is the size of the regular buffer at the end of Phase 1 at source node $Source^{(k)}$ (and ϕ' denotes the derivative of ϕ).

In order to evaluate potential decrease during phases 2 to 4, Awerbuch and Leighton use the following mechanism. They consider the potential decrease that would be induced in Phase 2 by a flow of value $(1+2\varepsilon)d_k$ for each commodity k. By assumption, such a flow exists, even if we do not know how to ship it. Since proposed algorithm decreases the flow locally for each edge as much as possible during phase 2, then phase 2 decreases the potential as much as what a $(1+2\varepsilon)d_k$ flow does. Using this argument, it can be proved (see [40], since the proof in the original paper contains one mistake) that the potential decrease during phases 2, 3 and 4 is at least

$$\left(1+\frac{3\varepsilon}{2}-\varepsilon^2\right)d_k\phi'_k(s_k)-\frac{\varepsilon(2+5\varepsilon)}{8n}.$$

Therefore, the overall drop in potential for all four phases is given by

$$\left(\frac{\varepsilon}{2}-\varepsilon^2\right)d_k\phi'_k(s_k)-\frac{\varepsilon(2+5\varepsilon)}{8n}.$$

We can then set the value of the constant in the O of equation (7) so that the previous equation is always positive. Therefore if any regular buffer is saturated at the end of Phase 1,

then the overall potential decreases during the whole round. Therefore, since regular buffers are not saturated at the end of the phase 1 of the first round, the overall potential OP during the whole process is bounded by

$$OP \leqslant \sum_k 2m\phi_k(Q_k) \leqslant \frac{2mK(2+5\varepsilon)}{\varepsilon(\frac{1}{2}-\varepsilon)}.$$

Therefore, the sum of the sizes of all regular and overflow buffers associated to commodity k can be bounded from the last equation and is of order

$$O\left(\frac{mnd_k\left(k+\ln(\frac{k}{\varepsilon})\right)}{\varepsilon}\right),$$

and therefore remains bounded during the whole algorithm. Thus, since $(1+\varepsilon)d_k$ units of flow for commodity k are injected in the system at each round, the throughput for flow of commodity k is arbitrarily close to d_k, what achieves the proof.

3.3.3 Adaption to the throughput optimization problem

As already noted, the maximization of throughput for k sets of independent tasks cannot be reduced to a classical multi-commodity flow problem, since under our model (one-port model for communications, one node processes several type of tasks), the capacity constraints are written for each node and not for each edge,

$$\begin{cases} \forall u \neq P_{\text{master}}, \mathit{Source}, \mathit{Sink} & \displaystyle\sum_k \sum_{v\in\Gamma(u)} \frac{sent_{u\to v}^{(k)} \cdot b^{(k)}}{b_{u,v}} \leqslant 1 \qquad \text{(one-port model at } P_u\text{)} \\ \forall u \neq P_{\text{master}}, \mathit{Source}, \mathit{Sink}, & \displaystyle\sum_k sent_{u\to \mathit{Sink}^{(k)}}^{(k)} \times c^{(k)} \leqslant c_u \quad \text{(processing constraint at } P_u\text{)} \end{cases}$$

Nevertheless, the main ideas of the Awerbuch-Leighton algorithm still apply to our problem. Let us consider a very simple adaptation of the algorithm, where only Phase 2 has been changed. Recall that the aim of Phase 2 is to minimize as much as possible the overall potential by exchanging tasks on the edges, given the capacity constraints. In our context, we need to change this minimization phase, given the set of constraints for both processing and outgoing communications (we do not take incoming communications into account since we assume a tree-shaped platform).

Let us consider node P_u. The sizes of buffers for tasks of type k at P_u are denoted $q_{u\to v}^{(k)}$ for the edge between P_u and P_v, where $P_v \in \Gamma(P_u)$ and by $q_{\mathit{Sink}^{(k)}}$ for the edge between P_u and $\mathit{Sink}^{(k)}$. We also denote by $p_{u\to v}^{(k)}$ the size of the buffer for tasks of type k at the tail of the edge between P_u and P_v. Because of last phase 3, the buffer for tasks of type k at $\mathit{Sink}^{(k)}$ is initially empty. We will denote by $f_{u\to v}^{(k)}$ the number of tasks of type k sent to P_v and by $\alpha_u^{(k)}$ the number of tasks of type k processed by P_u. Then, the corresponding minimization problems at P_u become

- For processing:

  $\textbf{Minimize} \sum_k \left(\exp\left(\gamma_k(q_{\mathit{Sink}^{(k)}} - \alpha_u^{(k)})\right) + \exp\left(\gamma_k \alpha_u^{(k)}\right)\right)$ (the new potential)
  Under the constraints
  $\begin{cases} \forall k, \quad \alpha_u^{(k)} \geqslant 0, & \text{(the number of processed tasks is non negative)} \\ \sum_k c^{(k)}\alpha_u^{(k)} \leqslant c_u, & \text{(time needed to process all types of tasks)} \end{cases}$

- For outgoing communications:

$$\textbf{Minimize} \sum_k \sum_{v\in\Gamma(u)} \left(\exp\left(\gamma_k(q_{u\to v}^{(k)} - f_{u\to v}^{(k)})\right) + \exp\left(\gamma_k(p_{u\to v}^{(k)} + f_{u\to v}^{(k)})\right)\right) \quad \text{(the new potential)}$$

Under the constraints

$$\begin{cases} \forall k, \quad f_{u\to v}^{(k)} \geqslant 0, & \text{(the number of transmitted tasks is non negative)} \\ \sum_k \sum_v \frac{b^{(k)} f_{u\to v}^{(k)}}{b_{u,v}} \leqslant 1, & \text{(time needed to send all types of tasks to all children)} \end{cases}$$

Both optimization problems obey to the following general formulation:

$$\textbf{Minimize } F(f) = \sum_{i=1}^{K} \left(\exp\left(\gamma_i(q_i - f_i)\right) + \exp\left(\gamma_i(p_i + f_i)\right)\right)$$

Under the constraints

$$\begin{cases} \forall i, \quad f_i \geqslant 0, \\ \sum_i w_i f_i \leqslant c, \end{cases}$$

Such problems can be solved using general techniques of non linear convex optimization and Karush-Kuhn-Tucker conditions [39]. Indeed, the above optimization problem is a convex optimization problem since the constraints define a convex domain and

$$\forall i, \frac{\partial^2 F}{\partial {f_i}^2} = \gamma_i^2 \left(\exp\left(\gamma_i(q_i - f_i)\right) + \exp\left(\gamma_i(p_i + f_i)\right)\right) > 0.$$

Therefore, Karush-Kuhn-Tucker conditions apply to the optimal solution:

$$\exists \lambda, \begin{cases} \forall i, \quad \frac{\partial F}{\partial f_i} - \lambda \frac{\partial}{\partial f_i}\left(c - \sum_j w_j f_j\right) = 0 \text{ if } f_i \neq 0 \\ \lambda \geqslant 0 \text{ and } \forall i, \quad f_i \geqslant 0 \\ \lambda(c - \sum w_i f_i) = 0 \end{cases}$$

which is equivalent to

$$\exists \lambda, \begin{cases} \forall i, \quad \gamma_i\left(-\exp\left(\gamma_i(q_i - f_i)\right) + \exp\left(\gamma_i(p_i + f_i)\right)\right) + \lambda w_i = 0 \text{ if } f_i \neq 0 \\ \lambda \geqslant 0 \text{ and } \forall i, \quad f_i \geqslant 0 \\ \lambda(c - \sum w_i f_i) = 0 \end{cases}$$

and finally

$$\exists \lambda, \begin{cases} \forall i, \quad \lambda = \frac{\gamma_i}{w_i}\left(\exp\left(\gamma_i(q_i - f_i)\right) - \exp\left(\gamma_i(p_i + f_i)\right)\right) \text{ if } f_i \neq 0 \\ \lambda \geqslant 0 \text{ and } \forall i, \quad f_i \geqslant 0 \\ \lambda(c - \sum w_i f_i) = 0 \end{cases} \tag{8}$$

The first constraints of (8) define a strong relation between each f_i and λ. Let us consider a given i for a moment. We can define

$$\widetilde{\lambda} : \begin{cases}]0, \frac{q_i - p_i}{2}] & \mapsto [0, x_i[\quad \text{(where } x_i = \frac{\gamma_i}{w_i}\left(\exp\left(\gamma_i q_i\right) - \exp\left(\gamma_i p_i\right)\right)) \\ f_i & \to \frac{\gamma_i}{w_i}\left(\exp\left(\gamma_i(q_i - f_i)\right) - \exp\left(\gamma_i(p_i + f_i)\right)\right) \end{cases}$$

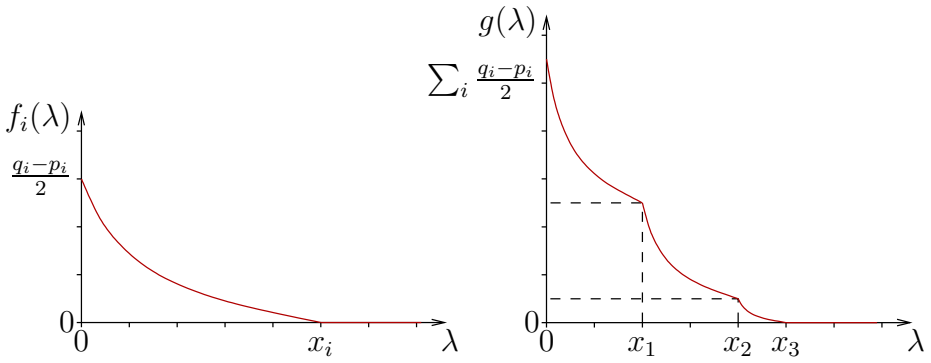


Figure 2: Relation between f_i and λ induced by the Karush-Kuhn-Tucker conditions.

We can easily check that $\widetilde{\lambda}'$ is negative. $\widetilde{\lambda}$ is thus strictly decreasing and is a bijection from $]0, \frac{q_i-p_i}{2}]$ to $[0, x_i[$. Therefore, we can also define

$$f_i : \begin{cases} [0,+\infty[& \mapsto [0, \frac{q_i-p_i}{2}] \\ \lambda & \to \begin{cases} \widetilde{\lambda}^{-1}(\lambda) = \frac{1}{\gamma_i} \ln \left(\frac{-\left(\frac{\lambda w_i}{\gamma_i}\right) + \sqrt{\left(\frac{\lambda w_i}{\gamma_i}\right)^2 + 4\exp(\gamma_i(p_i+q_i))}}{2\exp(\gamma_i p_i)} \right) & \text{if } \lambda < x_i \\ 0 & \text{if } \lambda \geqslant x_i \end{cases} \end{cases}$$

It follows that f_i is also a decreasing function of λ (see Figure 2). As a consequence, the following function g is also a decreasing function of λ:

$$g : \begin{cases} [0,+\infty[& \mapsto [0, \sum_i w_i \frac{q_i-p_i}{2}] \\ \lambda & \to \sum_i w_i f_i(\lambda) \end{cases}$$

Let us use the last constraint of system (8) to determine the solution of our optimization problem:

- If $c > g(0) = \sum w_i \frac{q_i-p_i}{2}$, then as g is a decreasing function of λ, the last condition of (8) holds true only if λ is equal to 0. As a consequence, the optimal solution of our minimization problem is found when $\lambda = 0$ and therefore we have $f_i = \frac{q_i-p_i}{2}$ for all i.
- If $c \leqslant \sum w_i \frac{q_i-p_i}{2}$, then we can find find λ such that $g(\lambda) = c$. Given the structure of g, this can be done by first determining the interval such that $g(x_j) \leqslant c$ and $g(x_{j+1}) < c$. Note that for $\lambda \in [x_j, x_{j+1}]$, the values of $f_1(\lambda), \ldots, f_j(\lambda)$ are all zeros. Then we can determine the exact value of λ such that $g(\lambda) = c$ by dichotomic search in $[x_j, x_{j+1}]$ (g being hard to formally inverse). From the value of λ we can then find the non-zeros values of f_i.

This provides a method to solve the optimization problems for outgoing communications under the one-port model and for processing several tasks at each node.

3.3.4 Hints for a distributed implementation

We have seen that it is possible to adapt the algorithm develop by Awerbuch and Leighton to our problem. Nevertheless, in the following, we will not present simulation results using our

modified version of Awerbuch Leighton algorithm. Indeed, several issues have to addressed before designing a fully distributed version of our algorithm. We provide here some ideas on how to use this method to derive a distributed algorithm to compute an optimal solution.

- First, the behavior of fictitious nodes and edges can be simulated in a distributed implementation: the master is in charge of all source nodes and corresponding edges, and each node with some processing capability is in charge of the simulation of the fictitious edges added to represent computation: as noted in the previous section, as in Phase 3, the buffer corresponding of task k at $Sink^{(k)}$ is emptied, this buffer is initially empty at the beginning of Phase 2.

- During Phase 1 of each round, we inject $(1+\varepsilon)d_k$ tasks of type k at source node k. A priori, this requires to know d_k and therefore the best achievable throughput! Nevertheless, there are several ways to circumvent this problem.

  1. We can solve the linear program once to get a rough evaluation of the throughput.
  2. We can also use the fundamental property of Awerbuch Leighton algorithm on the overall number of remaining tasks and use dichotomic search. Roughly, if we inject to much tasks at each round, then the platform will not be able to process them all, and therefore the overall number of remaining tasks in overflow buffer at source node will increase at each round. If the number of tasks in overflow buffers becomes larger than the theoretical bound on the size of the overflow queue derived in above section, then the amount of tasks injected at each round should be decreased. Unfortunately, the bounds known for the convergence rate of this process are very large, and may therefore not be of practical use.

- During the second phase of each round, a rational number of tasks is exchanged along each edge. Since we deal with tasks, we need to transfer integer number of tasks only. Indeed, we must ensure that the file associated to a given task is not spread amongst several processors! Nevertheless, there are again several possible ways to fix this problem.

  1. It is possible to solve the non linear multivariate linear optimization problem with rational numbers, and then to round obtained values to integers. Unfortunately, we did not manage to prove any approximation ratio for this process.
  2. We can also change the size of the round in order to obtain integer number of tasks, using lcm's, as we proposed in Section 3.2 to obtain a valid schedule from the solution of the linear program. nevertheless, this solution may lead to huge round periods and may therefore not be of practical use.

4 Design of Decentralized Heuristics

As shown in Section 3.2, given a tree-shaped platform and the set of all application parameters, we are able to compute an optimal periodic schedule. However, when trying to implement such a schedule, we face two main problems:

1. The period of the schedule (as computed in Section 3.2) is the lcm of the denominators of the solution of linear program (6). This period may then be huge, which entails

two embarrassing issues. First such a long period makes it potentially very difficult to adapt to load variations. Second, a long period implies sending large bunch of tasks at a time to a single processor, what incurs prohibitive memory overhead. It follows that we should try to implement dynamic heuristics that use the weights computed by the linear program (6) as hints to achieve a good throughput.

2. Computing those weights with a centralized algorithm (i.e. by solving linear program (6)) may become an issue when the size of the platform grows beyond a certain point. It may then be hard to collect up-to-date the information and inject them into the linear program. Moreover a slight variation of the data may result in a totally different solution. Indeed, even though the optimal value of the objective function changes smoothly with the input parameters, it may be the case that many different solutions have the same throughput and we could jump from one solution to the others. Consequently, we should try to design a decentralized computation of these weights and evaluate its quality regarding the optimal solution. Even though we have not been able to obtain any insight on the structure of the optimal solution for arbitrary trees, we will see that a few natural rules enable to get a decent decentralized computation of those weights.

In the following we consider four algorithms that make local decisions. We evaluate two different things:
- the impact of a decentralized control of the scheduling: how much do we loose when we use demand-driven heuristics compared to a tight periodic schedule built with the lcm of the solution of the linear program?
- the impact of a decentralized computation of the weights used in demand-driven heuristic.
Only the first heuristic below relies on centralized information, and it will serve as a reference to assess the performance of the four decentralized approaches.

4.1 Centralized demand-driven (*LP-BASED*)

If we allow for a centralized solution, we can solve the linear program (4) and dynamically control a demand-driven heuristic by feeding children only with tasks of type that they are assigned by the linear program. See Figure 3: each node records what has been sent to each child and applies the 1D-load balancing mechanism [8] when a child executes several task types.

This *demand-driven based on linear program* heuristic (*LP-BASED*) is expected to converge to the optimal throughput, but we have not been able to prove it. Even for a single application, i.e. in the pure bandwidth-centric case, there is much experimental evidence of this fact [33] but no proof is available. For several applications, we will see that in practice, the real throughput achieved by this heuristic is extremely close to optimal throughput.

However, to implement this heuristic, we need to know for each processor and each task type, (the number of tasks sent/computed per second. These rates can be computed using linear program (4) but this approach is not likely to scale very well. In other words, this heuristic does not satisfy our basic requirement of a decentralized solution.

4.2 First Come First Served

The *FCFS* heuristic is a purely demand-driven heuristic. All nodes keep sending requests to their parents, and parents fulfill them on a *First Come First Served* basis. The master

1: The local worker starts by requesting a few tasks.
2: **Loop**
3: Send requests to your father (and make sure that you do not have too much pending requests)
4: Select the type of application and the host (yourself or one of your child) using the 1D-load balancing mechanism.
5: Get incoming requests (from your local worker or one of your child) if any
6: Get incoming tasks (from your father) if any
7: **If** you have one task and a request that match the 1D-load balancing mechanism choice **Then**
8: Send a task to the corresponding host (respecting the 1-port constraint)
9: **Else**
10: Wait for a request or a task

Figure 3: Dynamic demand-driven 1D-load balancing algorithm

controls the fairness by alternately sending different task types, using the 1D load-balancing mechanism [8] with priority weights $1/w^{(k)}$. The 1D load-balancing mechanism works as follows: if n_k tasks of type k, $1 \leqslant k \leqslant K$, have already been sent by P_{master}, the next task to be sent will be of type ℓ, where

$$\frac{n_\ell + 1}{w^{(\ell)}} = \min_{1 \leqslant k \leqslant K} \frac{n_k + 1}{w^{(k)}}$$

Obviously, the choice of type ℓ achieves the best possible weighted load, given the tasks that have already been executed.

This is a simple scheduling scheme but also very natural. It will serve as a reference for more complex heuristics, which are duly expected to perform better!

4.3 Coarse-Grain

This heuristic (*CGBC*) builds upon our previous work for scheduling a single application on a tree shaped platform [9, 6]. In this framework, we have designed a decentralized algorithm, called *bandwidth-centric*. Each node only needs to know local information, i.e. the bandwidth and speed of its children. Based upon this information, a node selects the children that it will enroll to compute tasks. Typically, not all children will be enrolled, only those with larger bandwidth. Once the selection of children is made, the algorithm amounts to serve them on a demand-driven basis. In other words, the *bandwidth-centric* algorithm is a variant of *FCFS* where only a subtree is selected to take part to the computation.

The idea of the *coarse-grain* heuristic is to assemble several tasks into a large one. More precisely, we build a macro-task made out of $w^{(k)}$ tasks of type k, for each k. The computational weight of a macro-task is $W = \sum_{k=1}^{K} w^{(k)} \cdot c^{(k)}$, and its communication weight is $\Delta = \sum_{k=1}^{K} w^{(k)} \cdot b^{(k)}$. Thereafter, we allocate macro-tasks using a pure bandwidth-centric approach, which can be summarized as follows: suppose that the subtree rooted at node P_u as depth 1 (meaning that no children of P_u has itselft children). We sorts the children of P_u in the tree $P_{v_1}, P_{v_2}, \ldots, P_{v_i}$ such that $b_{u,v_1} \geqslant b_{u,v_2} \ldots \geqslant b_{u,v_i}$; only the first k children will be

enrolled in the demand-driven execution, where k is the smallest integer such that

$$\sum_{j=1}^{k} \frac{\Delta}{b_{u,v_j}} \cdot \frac{c_{v_j}}{W} > 1 \tag{9}$$

(and all children are enrolled if no such k exists). Intuitively, node P_u needs $\frac{\Delta}{b_{u,v_j}}$ time-units to send a task to P_{v_j}, which it will execute in $\frac{W}{c_{v_j}}$ time-units. Hence P_u spends $\frac{\Delta}{b_{u,v_j}} \cdot \frac{c_{v_j}}{W}$ time-units to keep P_{v_j} active for one time-unit. Within one time-unit, P_u can feed the first $k-1$ children at full rate, and the k-th child at a possibly reduced rate; but P_u should not send any task to the next children, if any.

The P_u computes its *aggregated computing speed* $c_{u,\,\mathrm{agg}}$ defined as the computing speed of a single node equivalent to the subtree rooted at P_u:

$$c_{u,\,\mathrm{agg}} = c_u + \sum_{j=1}^{k-1} c_{v_j} + \frac{\left(1 - \sum_{j=1}^{k-1} \frac{c_{v_j}}{W} \cdot \frac{\Delta}{b_{u,v_j}}\right) \cdot b_{u,v_k} \cdot W}{\Delta}$$

This speed is used in Equation 9 at the father of P_u, where P_u is considered as a child with computing speed $c_{u,\,\mathrm{agg}}$ and with no children. This bottom-up process leads to the knowledge of the global throughput of the platform at the master, and to the selection of the participating nodes.

The *coarse-grain* heuristic will not reach the optimal fair throughput, because some resources are not used at their best capacity. Indeed, we have seen (Proposition 1) that nodes with faster incoming links should process only tasks with larger communication-to-computation ratio). In fact it is very likely that in presence of a very communication-intensive macro-task fewer nodes will be enrolled by the bandwidth-centric scheduler; however, in the optimal solution, deeper nodes in the tree would probably process less demanding applications (those with a small communication-to-computation ratio).

4.4 Blind Co-Scheduling

The main idea of the *blind co-scheduling* heuristic (*BCS*) is to superpose the bandwidth-centric trees for each type of tasks and to run all of them in parallel. More precisely, there are K schedulers that run simultaneously.

Running many independent schedulers in parallel on the same resources will entail some problems for performance evaluation. In all our experiments we do enforce the one-port constraint for all schedulers. But here the for *blind co-scheduling* heuristic, we have not enforced this constraint globally across the schedulers (nor did we enforce the exclusive usage of a CPU, but this less likely to obviate results). We did not enforce any global constraint to simulate as accurately as possible the behavior of a blind co-scheduling. Therefore, in some sense, this heuristic is favored compared to the other ones as a node is then potentially able to make a better use of network resources (many outgoing communications could simultaneously occur at the same location).

However, in our experiments we have observed that this performance evaluation issue was not too much a difficulty and that bandwidth and CPU speed measurements evaluations stabilized rather quickly. As we will see, the main trouble with this approach results from the greedy sharing of resources among the different trees. Some trees are particularly unfavored compared to other ones, and the *blind co-scheduling* heuristic suffers from this problem.

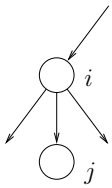
4.5 Decentralized Demand-Driven

This heuristic aims at implementing a decentralized version of the *LP-BASED* approach (Section 4.1). We seek for a decentralized demand-driven algorithm (*DATA-CENTRIC*) that converges, or comes close, to a solution of program (4), even if it is not optimal.

We sort the task types by non-increasing communication-to-computation ratio. Using a pure bandwidth-centric approach [9, 6], we start by computing the optimal rates for tasks of type 1. This load repartition being particularly unfair, we try to improve it by repeatedly performing some of the following operations (in the following, A (resp. B) denotes the application with currently has the highest (resp. lowest) throughput). All these operations are exclusive and are to be applied repeatedly in this order:

Communication Trading If a processor is partially idle but is receiving tasks of type A, its father exchanges some A tasks against some B tasks. It does this exchange so that its bus occupation remains the same though by trading communication time. Therefore, it lowers the throughput of type A tasks processed and increases the throughput of type B tasks while not changing the bandwidth saturation. It just leads to a more fair solution that makes better use of the computing resources. Note that this operation makes sense only if A has a higher communication-to-computation ratio than B.

Consider a processor j willing to trade ε_A tasks of type A against ε_B tasks of type B. Let us denote by CPU the cpu occupation of processor j :



$$CPU = \sum_k \frac{\alpha_j^{(k)}.c^{(k)}}{c_j}$$

The following constraints must hold true:

$$\varepsilon_A \leqslant \alpha_j^{(A)} \qquad (P_j \text{ cannot give more than it actually has}) \tag{10}$$

$$CPU - \varepsilon_A \frac{c^{(A)}}{c_j} + \varepsilon_B \frac{c^{(B)}}{c_j} \leqslant 1 \qquad (\text{cpu occupation has to be smaller than 1}) \tag{11}$$

Last, as we are trading communication time, we have:

$$\varepsilon_A \frac{b^{(A)}}{b_j} = \varepsilon_B \frac{b^{(B)}}{b_j} \tag{12}$$

Combining equations (11) and (12), we get:

$$\varepsilon_A \leqslant \frac{1 - CPU}{\frac{c^{(B)}}{c_j}.\frac{b^{(A)}}{b^{(B)}} - \frac{c^{(A)}}{c_j}}$$

Because we do not want to reduce too much the imbalance between $\alpha^{(A)}$ and $\alpha^{(B)}$, we add the following constraint:

$$\alpha^{(A)} - \varepsilon_A \geqslant \alpha^{(B)} + \varepsilon_B,$$

which leads to

$$\varepsilon_A \leqslant \frac{\alpha^{(A)} - \alpha^{(B)}}{1 + \frac{b^{(A)}}{b^{(B)}}}$$

Therefore, we have:

$$\varepsilon_A = \min\left(\alpha_j^{(A)}, \frac{1-CPU}{\frac{c^{(B)}}{c_j}.\frac{b^{(A)}}{b^{(B)}} - \frac{c^{(A)}}{c_j}}, \frac{\alpha^{(A)} - \alpha^{(B)}}{1 + \frac{b^{(A)}}{b^{(B)}}}\right) \quad \text{and} \quad \varepsilon_B = \frac{b^{(A)}}{b^{(B)}}\varepsilon_A$$

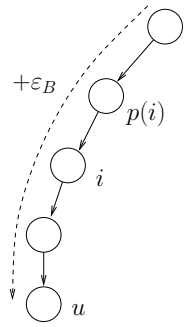
Gap filling It may be the case that some bandwidth is not used and that a remote processor P_u could receive more tasks of an unfavored application.

Let us denote by ε_B the amount of tasks of type B that this processor could handle. If we denote by CPU the cpu occupation of processor P_u, we have:

$$CPU = \sum_k \frac{\alpha_u^{(k)}.c^{(k)}}{c_u}$$

and the following condition on ε_B has to hold true:

$$CPU + \varepsilon_B \frac{c^{(B)}}{c_u} \leqslant 1$$



We also need to verify that there is enough free network along the path from the root node to P_u. Therefore for any node i along this path, we need the following condition on ε_B to hold true:

$$\underbrace{\sum_k \sum_j \frac{sent_{p(i)\to j}^{(k)}.c^{(k)}}{c_j} + \varepsilon_B \frac{b^{(B)}}{b_i}}_{bus\_occupation(p(i))} \leqslant 1$$

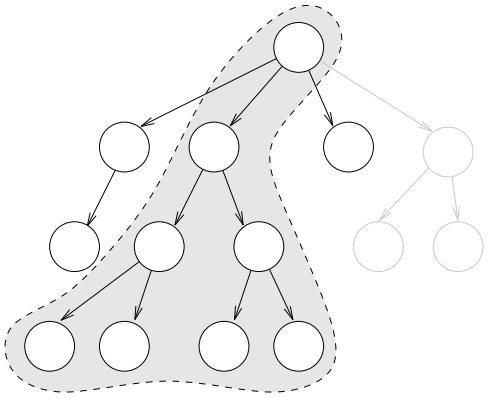
Last, we do not want to reduce too much the imbalance between $\alpha^{(A)}$ and $\alpha^{(B)}$, we add the following constraint:

$$\alpha^{(A)} \geqslant \alpha^{(B)} - \varepsilon_B.$$

Therefore, we have:

$$\varepsilon_B = \min\left(\frac{c_u(1-CPU)}{c^{(B)}}, \alpha^{(B)} - \alpha^{(A)}, \min_{\substack{i \in \text{ path from} \\ \text{the root to } P_u}} \left(\frac{1 - bus\_occupation(p(i))}{b^{(B)}}.b_i\right)\right)$$

Bus de-saturation The bus may have been saturated by tasks with a high communication-to-computation ratio. We may then still be using only workers with high communication capacity. In such a situation, the tree has to be *widened* and the only way to do that is to reduce the amount of tasks of type A that are processed by the subtrees. The $\alpha_i^{(A)}$ and $sent_{i\to j}^{(A)}$ values of any node of the branch with the smallest bandwidth that process some tasks of type A are then scaled down by a factor of 0.9. This operation allows to decrease the communication resource utilization and precedes *"Gap filling"* operations.



Task trading on the master At some point (when application A is processed only on the root node) we may have no choice but to change what the master is doing. Let us consider that the master will trade ε_A tasks of type A against ε_B tasks of type B. Then we will have the following constraints:

$$\begin{aligned} \varepsilon_A &\leqslant \alpha^{(A)}_{root} \\ \alpha^{(A)} - \varepsilon_A &\geqslant \alpha^{(B)} + \varepsilon_B \\ \varepsilon_B . \frac{c^{(B)}}{c_{root}} &= \varepsilon_A . \frac{c^{(A)}}{c_{root}} \end{aligned}$$

Therefore, we have

$$\varepsilon_A = \min\left(\alpha^{(A)}_j, \frac{\alpha^{(A)} - \alpha^{(B)}}{1 + \frac{b^{(A)}}{b^{(B)}}}\right) \quad \text{and} \quad \varepsilon_B = \frac{b^{(A)}}{b^{(B)}}\varepsilon_A$$

Those operations are continuously performed (with the listed order of precedence) until we reach a satisfying balance like:

$$\frac{\alpha^{(k_{\max})} - \alpha^{(k_{\min})}}{\alpha^{(k_{\min})}} < 0.05$$

We use those weights to dynamically control (using the algorithm of Figure 3) the task distribution.

5 Simulation Results

5.1 Evaluation methodology

In this section we describe in a comprehensive way the experimental framework that we have set up to assess the efficiency of our heuristics.

5.1.1 Throughput evaluation

Determining that we have reached a steady-state and computing the throughput of a schedule are straightforward problems when one considers periodic schedules. These are however much trickier problems when the schedule is not periodic. Consider for example a heuristic that first process all tasks of application 1, then all tasks of application 2, and so on. At a global scale, no steady state has actually occurred yet. At a local scale, though, one can observe K different steady-state periods. How could we define the throughput for each of these applications? It could be the total number of tasks of each application divided by the makespan of the whole schedule. However, this would not really reflect the fairness of the schedule. We would rather say that a schedule like the one we just described achieves a fair throughput (the quantity we aim at maximizing) equal to 0.

In fact, once all tasks of a given application have been processed, the situation is completely different. Indeed being fair to this application does not make sense anymore. That is why we should only consider the period where tasks of any application are still in the system to compute the fair throughput. Let us denote by T the first date where all tasks of an

```
1: Push(nodes_to_expand, root)
2: While root = Shift(nodes_to_expand) Do
3:   If n = 0 Then
4:     Return {there is no more node to add}
5:   Repeat {to avoid premature pruning}
6:     nb_child ← rand(0..degree_max)
7:     If nb_child ⩾ n Then
8:       nb_child ← n
9:   Until |nodes_to_expand| > 0 or nb_child > 0
10:  For u ∈ {1 ... nb_child} Do
11:    n ← n − 1
12:    Create a new_node connected to root
13:    Push(nodes_to_expand, new_node)
```

Figure 4: Tree generation

application have been processed. Let us also denote by $N_k(t)$ the number of tasks of type k that have been processed in time period $[0, t]$. We can then define the achieved throughput ρ_k for application k by:

$$\rho_k = \frac{N_k((1-\varepsilon)T) - N_k(\varepsilon T)}{(1-2\varepsilon)T} \quad \text{, where } \varepsilon \in]0, 0.5[.$$

The epsilon is just a measurement artifact to get rid of initial and final instabilities (in practice, we set ε to be equal to 0.1). In the following, we will refer to ρ_k as the *experimental throughput* of application k as opposed to the expected throughput that can be computed solving linear program (6). Like wise, the minimum of the weighted expected throughput will be referred as *experimental fair throughput*. For some heuristics (e.g. *LP-BASED*, *DATA-CENTRIC* and *CGBC*) we can easily compute an expected theoretical throughput. We will see in section 5.2.1 how implementation issues can affect the deviation of the experimental fair throughput from the expected theoretical fair throughput.

5.1.2 Platform generation

The platforms used in our experiments are random trees described by two parameters: the number of nodes n and the maximum degree $degree_{max}$. We use the algorithm of Figure 4 to generate the interconnection network topology. This breadth-first growth enables to have wide trees rather than filiform ones. In our experiments, we have generated trees of 5, 10, 20, 50 and 100 nodes. The maximum degree was equal to 2, 5, or 15 and 10 trees of each configuration have been generated (i.e our platform set comprised 150 trees in total).

Then we assign typical capacity, latency and CPU power values on edges and nodes at random. Those values come from real measurements performed on machines spread across the internet with tools like pathchar. CPU power ranges from 22.151 Mflops (an old Pentium Pro 200MHz) to 171.667 Mflops (an Athlon 1800). Capacity ranges from 110 kb/s to 7 Mb/s and latency ranges from 6 ms to 10 s. Note that in the simulator that we are using (see Section 5.1.4), latency is a limiting factor as well as the link capacity for determining the effective bandwidth of a connection.

5.1.3 Application generation

An application type being mainly characterized by its *communication-to-computation ratio* (*CCR*), a set of different application types is described by the interval in which the *CCR* of the applications lie. For example suppose we consider two competing applications. If the tasks of the first (resp. second) application require 21.739 Mflop and 100 Mb (resp. 100,000 Mflop and 100 Mb), the *CCR* is roughly equal to 4.6 (resp. 0.001). To fix the orders of magnitude, a *CCR* of 4.6 is typical of a two 3500×3500 matrix addition and a *CCR* of 0.001 is typical of a two 3500×3500 matrix multiplication. It is not reasonable to consider applications with a *CCR* larger than 4.6 because a matrix addition is an operation where the amount of computations is very small regarding the amount of data to communicate. The *CCR* interval is therefore equal to $[CCR_{min}, CCR_{max}] = [0.001, 4.6]$. When we generate application sets with more than two types of tasks, we ensure that the *CCR* are evenly distributed in the *CCR* interval, which enables us to completely characterize an application set by the corresponding *CCR* interval. In our simulations we always have set CCR_{min} to be equal to 0.001 (i.e. there is an highly parallel application that is likely to take advantage of the whole distributed platform) and CCR_{max} ranges from 0.002 to 4.6.

5.1.4 Heuristic implementation

The experiments described herein have been performed using the SimGrid simulator [34]. The simulated platform is therefore much more complex than the platform model used to design our heuristics. Therefore c_i and b_i values are continuously measured from within the simulator and used as such to compute the weights used in the algorithms of Section 4. As explained in section 4.5, most heuristics rely on a dynamic demand-driven 1D-load balancing algorithm (see Figure 3). This algorithm heavily relies on a request mechanism. Each of these requests account for a few bytes that simply tell a processor that a child needs some more tasks. This request mechanism has been fully implemented and we have ensured that no deadlock occurred in our thousands of experiments, even when some load-variations occurred, resulting in brutal weight modifications. Last, throughput evaluation has been performed in most experiments by running 200 tasks per application.

5.2 Case study

5.2.1 Impact of a decentralized control of the scheduling

As we use a decentralized control of the scheduling, the experimental fair throughput may be smaller than the expected theoretical one (that would be reached with a tight periodic schedule built with the lcm of the values returned by the linear program). In particular, one reason why we may not reach the optimal throughput is that we somehow limit the number of requests (and therefore the amount of tasks stored on each node, even if this is not a strict constraint) that a node can send to its parents. Having a strict limit is hard because as we strictly enforce the 1D-load balancing, it may lead to deadlocks. Figure 5 depicts the experimental fair throughput deviation from the expected theoretical throughput for heuristics *CGBC*, *LP-BASED* and *DATA-CENTRIC* (computing the theoretical throughput of the other heuristics being out of the scope of this article) when each node maintains a maximum number of 10 pending requests in steady-state.

The average deviation is equal to 9.426%. When we increase the maximum number of pending requests and the buffer size by a factor ten, the mean average deviation drop to 0.334%

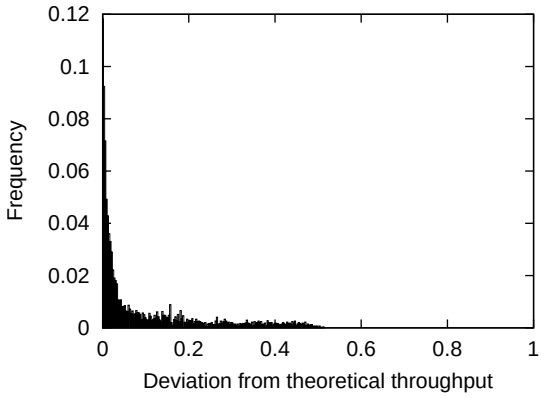


Figure 5: Deviation of experimental fair throughput from expected theoretical throughput

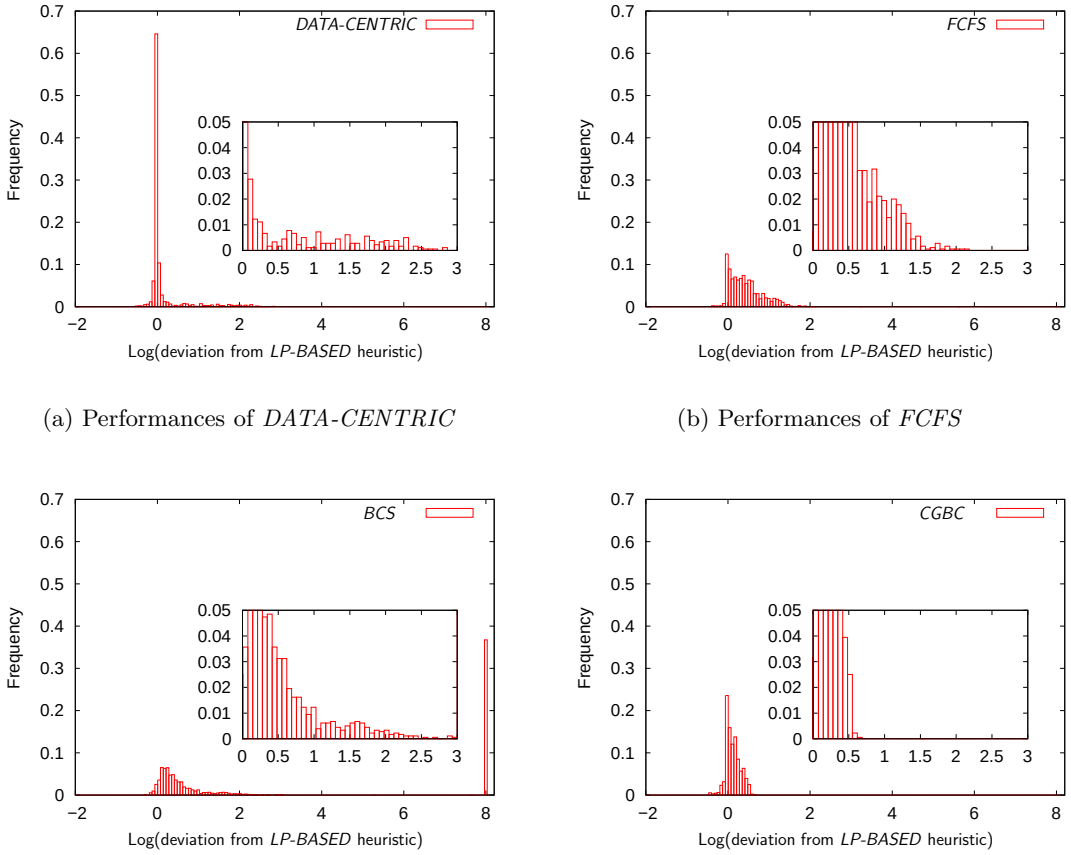
(this throughput has been evaluated with 10 times more tasks). Note that this problem of buffer size has already been pointed out in [21, 12]. Finding the optimal throughput when buffer sizes are bounded is a strongly NP-hard problem even in very simple situations [12]. Even though increasing buffer size leads to much better throughput, we consider that this is not a viable approach. As a consequence, we will only consider in the following the case where each node maintains a maximum number of 10 pending requests in steady-state.

5.2.2 Heuristic performances

Let us first compare the relative performances of our five heuristics (*FCFS*, *BCS*, *CGBC*, *LP-BASED* and *DATA-CENTRIC*). More precisely, for each experimental setting (i.e. a given platform and a given *CCR* interval), we compute the logarithm of the ratio of the experimental fair throughput of *LP-BASED* with the experimental fair throughput of a given heuristic (applying a logarithm enables us to have a symmetrical value). Therefore, a positive value means that *LP-BASED* performed better than the other heuristic. Figure 6 depicts the histogram plots of these values.

First of all, we can see that most values are positive, which assesses the superiority of *LP-BASED*. Then, we can see on Figure 6(a) that *DATA-CENTRIC* is most of the time very close to *LP-BASED*, despite the distributed computation of the weights. However, the geometric average of these ratios is equal to 1.164, which is slightly larger than the geometric average for *CGBC* (1.156). The reason is that even though in most settings *DATA-CENTRIC* ends up with a very good solution, in a few ones its performances are very bad (up to 16 times worse than *LP-BASED*). On the opposite, *CGBC* (see Figure 6(d)) is much more stable since its worst performance is only two times worse than *LP-BASED*. Note that those failures happen on any type of tree (small or large, filiform or wide) and that the geometric average of these two heuristics are always very close to each other. We also have checked that these failures are not due to an artifact of the decentralized control of the scheduling by ensuring that the theoretical throughput has the same behavior. We are still investigating the reasons why *DATA-CENTRIC* fails on some instances and suspect that it is due to the use of the sometimes misleading intuition of Proposition 1.

Unsurprisingly, *BCS* leads to very bad results. In many situations (more than 35% actually), an application has been particularly unfavored and the fair experimental throughput was close to 0. The logarithm of the deviation for these situations has been normalized to



(a) Performances of *DATA-CENTRIC*

(b) Performances of *FCFS*

(c) Performances of *BCS*

(d) Performances of *CGBC*

Figure 6: Logarithm of the deviation from *LP-BASED* performances

8. These poor results advocate the need for fairness guarantees in distributed computing environment like the ones we consider.

Last, the geometrical average of *FCFS* is 1.564 and in the worst case, its performances are more than 8 times worse the *LP-BASED* situation. On the average, *FCFS* is therefore much worse than *LP-BASED*. On small platforms, the performances for *FCFS* and *CGBC* have the same order of magnitude. However, on larger ones (50 and 100), *CGBC* performs much better (geometrical average equal to 1.243) than *FCFS* (geometrical average equal to 2.0399).

6 Related Work

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

Models for heterogeneous platforms – In the literature, one-port models come in two variants. In the unidirectional variant, a processor cannot be involved in more than one communication at a given time-step, either a send or a receive. In the bidirectional model, a processor can send and receive in parallel, but at most to a given neighbor in each direction. In both variants, if P_u sends a message to P_v, both P_u and P_v are blocked throughout the communication.

The bidirectional one-port model is used by Bhat et al. [17, 18] for fixed-size messages. They advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously". Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network". Experimental evidence of this fact has recently been reported by Saif and Parashar [43], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi et al. [4] Liu [35] and Khuller and Kim [32]. In this simpler model, the communication time only depends on the sender, not on the receiver: in other words, the communication speed from a processor to all its neighbors is the same.

Finally, we note that some papers [5, 7] depart form the one-port model as they allow a sending processor to initiate another communication while a previous one is still ongoing on the network. However, such models insist that there is an overhead time to pay before being engaged in another operation, so there are not allowing for fully simultaneous communications.

Steady-state scheduling – The steady-state approach has been pioneered by Bertsimas and Gamarnik [14]. This technique has been used in [9, 6] to schedule independent tasks on heterogeneous master-slave platforms. This is exactly the problem dealt with in this paper, but for a single application. Bandwidth-centric trees are introduced in [9], and extensive experiments are reported in [33]. Autonomous protocols for bandwidth-centric scheduling are investigated in [21].

The steady-state approach has been used by Hong et al. [30] who extend the work in [9] to deploy a divisible workload on a heterogeneous platform. Thanks to a special model,

it becomes possible to apply powerful graph techniques on the target platform (a graph weighted by link capacities and processor speeds): the solution is then a maximal flow in this graph. In this new model, each communication link is labelled with a value $l_{i,j}$ that represents the number of data which may be transferred through this link in one time-unit. Besides, the processors have a limited communication capacity: P_i can send at most b_i^{in} data and receive at most b_i^{out} data in one time-unit. This particular model can be justified by a network interface limited capacity at node P_i, whereas the same node P_i is able to open as many connections as it wants (one per neighbor). Contrarily to all other models, the one-port model is not assumed here, but the total capacity of all ports is limited. In [30], Hong et al. propose a decentralized algorithm to solve the problem of finding a maximum flow, and thus to coordinate the resources in the platform.

Scheduling divisible loads – Divisible load applications can be divided among worker processes arbitrarily, i.e. into any number of independent pieces. This corresponds to a perfectly parallel job: any sub-task can itself be processed in parallel, and on any number of workers. In practice, this model is an approximation of applications that consist of large numbers of identical, low-granularity computations. Because the applications considered in this paper are composed of independent tasks, they are not fully divisible: their granularity is fixed by the size of a task. However, allowing for scheduling rational number of tasks in stead-state mode makes the problem very similar to divisible-load scheduling. The main difference comes from the organization of the communications. Instead of scheduling several communications, one for each task, the divisible load approach consists in scheduling a single communication at the beginning of the operation. The cost of this communication is proportional to the amount of computation performed. This approach is termed *one-round* in the divisible load literature. A star graph is targeted in [46], with homogeneous links and different-speed processors. The extension to heterogeneous links is dealt with in [22]. See also [42, 1] for sharing bag of tasks in heterogeneous clusters, using a more refined platform model (including a detailed analysis of communication times).

The one-round approach is no the only one in the divisible literature; *multi-round* strategies divide the work into several chunks, whose processing is pipelined in several successive rounds. When successive rounds are identical, the model becomes quite close to steady-state scheduling. Indeed, the linear programming approach has successfully been applied to multi-round divisible load scheduling, either for a single application [11] or for several ones [36, 37].

Master-slave on the computational grid – Master-slave scheduling on the grid can be based on a network-flow approach [45, 44] or on an adaptive strategy [28]. Note that the network-flow approach of [45, 44] 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 [29]. Enabling frameworks to facilitate the implementation of master-slave tasking are described in [25, 47].

Fairness – Fairness is a classical criteria in network bandwidth allocation. Optimizing the sum of the throughputs is known as maximizing the throughput or *utility* of a network. Optimizing this kind of objective is natural for an access provider who receives an amount of money proportional to the throughput that he/she is able to provide and

who wants to maximize his profit. However, this criteria is known to be particularly unfair and leads to starvation. That is why in section 2.3, we choose to maximize the minimum of $\frac{\alpha^{(k)}}{w^{(k)}}$ rather than the sum. This criteria is known in the literature as *max-min* and is intuitively as fair as possible since all throughput are computed to be as close as possible from each other. Between these two extreme criteria, other criteria can be found (e.g. *proportional fairness* that maximize $\sum \frac{\log(\alpha^{(k)})}{w^{(k)}}$ or *minimum potential delay* that minimize $\sum \frac{1}{w^{(k)}\alpha^{(k)}}$) and provide nice alternatives. In fact all these criteria (utility, proportional fairness and minimize potential delay) amount to maximize the arithmetic, geometric and harmonic mean of the throughput [20]. It is a well-known fact in the network community [38] that max-min fairness is generally achieved by explicit-rate calculation (e.g. in ATM networks) and rather hard to achieve in a fully-decentralized fashion. Yet, fully distributed algorithms are known to realize proportional fairness (some variants of TCP). Adapting such algorithms to our framework is however really challenging as communications and computations are involved.

Last, in our framework, we use a global approach to ensure fairness. There is only one decision maker (the root of the tree) that optimizes the expected performance of the entire system over all jobs (by explicitly computing the rates). As we have seen with the *BCS* heuristic, non-cooperative approaches where each application optimizes its own throughput lead to a particularly unfair Nash equilibrium [31, 24]. A "third path" could be a cooperative approach where several decision makers (each of them being responsible for a given application) cooperate in making the decisions such that each of them will operate at its optimum. This situation can be modelled as a cooperative game like in [27, 26]. However in our situation, hierarchical resource sharing is rather hard to model.

7 Conclusion

In this paper, we present several heuristics for scheduling multiple applications on a tree-connected platform made of heterogeneous processing and communication resources. We first presented a centralized algorithm which, given the performances of all resources, computes an optimal schedule with respect of throughput maximization.

However, using the previous centralized algorithm requires to gather all the information about the platform at a single location, which may be unrealistic for a large scale distributed platform, whose parameters (bandwidths, processing power) are expected to evolve very quickly. We have therefore concentrated on distributed algorithms and designed several decentralized heuristics. The results obtained by the most sophisticated heuristics are very satisfying in practice, with respect of optimal throughput (as computed thanks to the optimal centralized algorithm). Nevertheless, it seems very hard to prove approximation ratios and to extend the results to more general platforms (whose topology is described as a general graph and not a tree).

We have also presented an adaption of the Awerbuch-Leighton algorithm for multi-commodity flows to our problem. Although several implementation problems have still to be solved, this approach represents a promising way both for proving approximation results and for considering more general platforms.

We believe that this paper constitutes an important contribution to the design of scheduling algorithm in the context of large-scale distributed and dynamic platforms.

References

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[2] Baruch Awerbuch and Tom Leighton. A simple local-control approximation algorithm for multicommodity flow. In *FOCS '93: Proceedings of the 24th Conference on Foundations of Computer Science*, pages 459–468. IEEE Computer Society Press, 1993.

[3] Baruch Awerbuch and Tom Leighton. Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks. In *STOC '94: Proceedings of the 26h ACM symposium on Theory of Computing*, pages 487–496. ACM Press, 1994.

[4] 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.

[5] 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.

[6] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15(4):319–330, 2004.

[7] Amotz Bar-Noy, Sudipto Guha, Joseph (Seffi) Naor, and Baruch Schieber. Message multicasting in heterogeneous networks. *SIAM Journal on Computing*, 30(2):347–358, 2000.

[8] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Trans. Computers*, 50(10):1052–1070, 2001.

[9] 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.

[10] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2004*. IEEE Computer Society Press, 2004.

[11] O. Beaumont, A. Legrand, and Y. Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Computing*, 29:1121–1152, 2003.

[12] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Independent and divisible tasks scheduling on heterogeneous star-schaped platforms with limited memory. In *PDP'2005, 13th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 179–186. IEEE Computer Society Press, 2005.

[13] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

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

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

[16] V. Bharadwaj, D. Ghose, and T.G. Robertazzi. Divisible load theory: a new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[17] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

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

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

[20] Thomas Bonald and Laurent Massoulié. Impact of fairness on internet performance. In *SIGMETRICS/Performance*, pages 82–91, 2001.

[21] 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.

[22] S. Charcranoon, T.G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on computers*, 49(9):987–991, September 2000.

[23] Einstein@Home. http://einstein.phys.usm.edu.

[24] Ferenc Forgó, Jenö, and Ferenc Szdarovsky. *Introduction to the Theory of Games: Concepts, Methods, Applications*. Kluwer Academic Publishers, 2 edition, 1999.

[25] 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.

[26] Daniel Grosu and Thomas E. Carroll. A strategyproof mechanism for scheduling divisible loads in distributed systems. In IEEE Computer Society Press, editor, *Proc. of the 4th International Symposium on Parallel and Distributed Computing (ISPDC 2005)*, 2005.

[27] Daniel Grosu, Antony T. Chronopoulos, and Michael Y. Leung. Load balancing in distributed systems: An approach using cooperative games. In IEEE Computer Society Press, editor, *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 501–510, 2002.

[28] 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.

[29] 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.

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

[31] John F. Nash Jr. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences USA*, 36:48–49, 1950.

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

[33] B. Kreaseck. *Dynamic autonomous scheduling on Heterogeneous Systems*. PhD thesis, University of California, San Diego, 2003.

[34] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

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

[36] Loris Marchal, Yang Yang, Henri Casanova, and Yves Robert. A realistic network/application model for scheduling divisible loads on large-scale platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2005*. IEEE Computer Society Press, 2005.

[37] Loris Marchal, Yang Yang, Henri Casanova, and Yves Robert. Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms. *Int. Journal of High Performance Computing Applications*, 2006, to appear.

[38] Laurent Massoulié and James Roberts. Bandwidth sharing: Objectives and algorithms. *Transactions on Networking*, 10(3):320–328, june 2002.

[39] Anthony L. Peressini, Francis E. Sullivan, and J.J. Jr. Uhl. *The Mathematics of Nonlinear Programming*. Springer, 1 edition, 1993.

[40] Yuval Rabani. Local competitive routing and the multi-commodity flow problem. Class notes, lectures 8-9: see http://www.cs.technion.ac.il/~rabani/236604.96.wi.html.

[41] T.G. Robertazzi. Ten reasons to use divisible load theory. *IEEE Computer*, 36(5):63–68, 2003.

[42] A. L. Rosenberg. Sharing partitionable workloads in heterogeneous NOws: greedier is not better. In *Cluster Computing 2001*, pages 124–131. IEEE Computer Society Press, 2001.

[43] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[44] 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.

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

[46] J. Sohn, T.G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on parallel and distributed systems*, 9(3):225–234, March 1998.

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

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399