FIFO scheduling of divisible loads with return messages under the one-port model

Olivier Beaumont, Loris Marchal, Veronika Rehn, Yves Robert

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

FIFO scheduling of divisible loads with return messages under the one-port model

Olivier Beaumont — Loris Marchal — Veronika Rehn — Yves Robert

N° 5738

October 2005

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5738--FR+ENG

FIFO scheduling of divisible loads with return messages under the one-port model

Olivier Beaumont, Loris Marchal, Veronika Rehn, Yves Robert

Thème NUM — Systèmes numériques
Projet Graal

Rapport de recherche n° 5738 — October 2005 — 25 pages

Abstract: This paper deals with scheduling divisible load applications on star networks, in presence of return messages. This work is a follow-on of [7, 8], where the same problem was considered under the 2-port model, where a given processor can simultaneously send and receive messages. Here, we concentrate on the one-port model, where a processor can either send or receive a message at a given time step. The problem of scheduling divisible load on star platforms turns out to be very difficult as soon as return messages are involved. Unfortunately, we have not been able to assess its complexity, but we provide an optimal solution in the special (but important) case of FIFO communication schemes. We also provide an explicit formula for the optimal number of load units that can be processed by a FIFO ordering on a bus network. Finally, we provide a set of MPI experiments to assess the accuracy and usefulness of our results in a real framework.

Key-words: Scheduling, divisible load, master-worker platform, heterogeneous cluster, return messages, one-port model

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 FIFO de tâches divisibles avec messages de retour sous le modèle un-port

Résumé : Dans ce rapport, nous nous intéressons à l'ordonnancement de tâches divisibles sur des plates-formes maître-esclave hétérogènes, en prenant en compte les messages de retour. Ce travail fait suite à [7, 8], où le problème était considéré avec le modèle 2-port, autorisant un processeur à envoyer et recevoir des données simultanément. Ici, nous nous intéressons au modèle un-port selon lequel ces deux opérations doivent être séquentialisées. Ordonnancer des tâches divisibles se révèle difficile dès que l'on prend en compte les messages de retour. Nous n'avons pas réussi à établir la complexité du problème général, mais nous proposons une solution optimale dans le cas particulier (mais important) des schémas de communication FIFO. Nous proposons également une formule pour calculer le débit optimal quand les liens de communication sont homogènes. Enfin, nous fournissons un jeu d'expériences utilisant MPI pour évaluer la précision et l'utilité de nos résultats dans un contexte réel.

Mots-clés : Ordonnancement, tâches divisibles, plate-forme maître-esclave, grappe hétérogène, messages de retour, modèle un-port

1 Introduction

This paper deals with scheduling divisible load applications on heterogeneous platforms. As their name suggests, divisible load applications can be divided among worker processors 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, the *Divisible Load Scheduling* model, or DLS model, is an approximation of applications that consist of large numbers of identical, low-granularity computations.

Quite naturally, we target a master-worker implementation where the master initially holds (or generates data for) a large amount of work that will be executed by the workers. In the end, output results will be returned by the workers to the master. Each worker has a different computational speed, and each master-worker link has a different bandwidth, thereby making the platform fully heterogeneous. The scheduling problem is first to decide how many load units the master sends to each worker, and in which order. After receiving its share of the data, each worker executes the corresponding work and returns the results to the master. Again, the ordering of the return messages must be decided by the scheduler.

The DLS model has been widely studied in the last several years, after having been popularized by the landmark book [10]. The DLS model provides a practical framework for the mapping of independent tasks onto heterogeneous platforms, and has been applied to a large spectrum of scientific problems. From a theoretical standpoint, the success of the DLS model is mostly due to its analytical tractability. Optimal algorithms and closed-form formulas exist for important instances of the divisible load problem. A famous example is the closed-form formula given in [5, 10] for a bus network. The hypotheses are the following: (i) the master distributes the load to the workers, but no results are returned to the master; (ii) a linear cost model is assumed both for computations and for communications (see Section 2.1); and (iii) all master-worker communication links have same bandwidth (but the workers have different processing speeds). The proof to derive the closed-form formula proceeds in several steps: it is shown that in an optimal solution: (i) all workers participate in the computation, then that (ii) they never stop working after having received their data from the master, and finally that (iii) they all terminate the execution of their load simultaneously. These conditions give rise to a set of equations from which the optimal load assignment α_i can be computed for each worker P_i.

Extending this property to a star network (with different master-worker link bandwidths), but still (1) without return messages and (2) with a linear cost model, has been achieved only recently [6]. The proof basically goes along the same steps as for a bus network, but the main additional difficulty was to find the optimal ordering of the messages from the master to the workers. It turns out that the best strategy is to serve workers with larger bandwidth first, independently of their computing power.

The next natural step is to include return messages in the picture. This is very important in practice, because in most applications the workers are expected to return some results to the master. When no return messages are assumed, it is implicitly assumed that the size of the results to be transmitted to the master after the computation is negligible, and hence has no (or very little) impact on the whole DLS problem. This may be realistic for some particular DLS applications, but not for all of them. For example suppose that the master is distributing files to the workers. After processing a file, the worker will typically return results in the form of another file, possibly of shorter size, but still non-negligible. In some situations, the size of the return message may even be larger than the size of the original message: for instance the master initially scatters instructions on some large computations to be performed by each worker, such as the generation of several cryptographic keys; in this case each worker would receive a few bytes of control instructions and would return longer files containing the keys.

Because it is very natural and important in practice, several authors have investigated the problem with return messages: see the papers [4, 15, 24, 3, 1]. However, all the results obtained so far are very partial. Intuitively, there are hints that suggest that the problem with return messages is much more complicated. The first hint lies in the combinatorial space that is open for searching the best solution. There is no reason for the ordering of the initial messages sent by the master to be the same as the ordering for the messages returned to the master by the workers after the execution. In some situations a FIFO strategy (the worker first served by the master is the first to return results, and so on) may be preferred, because it provides a smooth and well-structured pipelining scheme. In [1], Adler, Gong and Rosenberg show that all FIFO strategies are equally performing on a bus network, but even the analysis of FIFO strategies is a difficult open problem on a star network.

This work is a follow-on of [7, 8] where we have studied FIFO strategies under the *two-port* model, where the master can simultaneously send data to one worker and receive from another. In this paper, we study FIFO strategies under the *one-port* model, where the master can only be enrolled in a single communication at any time-step. The *one-port* model turns to be be more complicated to analyze, because of the additional constraint imposed on the communication medium. However, it is also more realistic in practice, and all the MPI experiments reported in Section 5 obey this model.

Adding return messages dramatically complicates the search for an optimal solution, despite the simplicity of the linear cost model. In fact, we show that the best FIFO schedule may well not involve all processors, which is in sharp contrast with previous results from the literature.

The main contributions of this paper are the characterization of the best FIFO strategy on a star network, together with an experimental comparison of them. Rather than simulations [7, 8] we perform extensive MPI experiments on heterogeneous platforms.

The rest of the paper is organized as follows. In Section 2, we state precisely the DLS problem, with all application and platform parameters. Section 3 deals with the characterization of the optimal FIFO solution. When the platform reduces to a bus, we are able to provide an explicit formula for the best throughput, as shown in Section 4. We report extensive MPI experiments in Section 5. Section 6 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 7.

2 Framework

2.1 Target platform and model

As illustrated in Figure 1, we target a *star network* $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$, composed of a master P_0 and of p workers P_i, $1 \leq i \leq p$. There is a communication link from the master P_0 to each worker P_i. In the *linear cost* model, each worker P_i has a (relative) computing power w_i: it takes $X.w_i$ time units to execute X units of load on worker P_i. Similarly, it takes $X.c_i$ time units to send the initial data needed for computing X units of load from P_0 to P_i, and $X.d_i$ time units to return the corresponding results from P_i to P_0. Without loss of generality we assume that the master has no processing capability (otherwise, add a

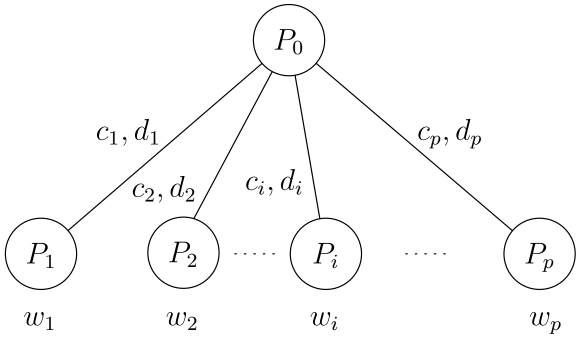


Figure 1: Platform

fictitious extra worker paying no communication cost to simulate computation at the master). Note that a *bus network* is a star network such that all communication links have the same characteristics: $c_i = c$ and $d_i = d$ for each worker P_i, $1 \leq i \leq p$.

It is natural to assume that the quantity $\frac{d_i}{c_i}$ is a constant z that depends on the application but not on the selected worker. In other words, workers who communicate faster with the master for the initial message will also communicate faster for the return message. In the following, we keep using both values d_i and c_i, because many results are valid even without the relation $d_i = zc_i$, and we explicitly mention when we use this relation.

The standard *one-port* model in DLS problems for communications is defined as follows:

- the master can only send data to, and receive data from, a single worker at a given time-step,
- a given worker cannot start execution before it has terminated the reception of the message from the master; similarly, it cannot start sending the results back to the master before finishing the computation.

In fact, this *one-port* model naturally comes in two flavors with return messages, depending upon whether we allow the master to simultaneously send and receive messages or not. If we do allow for simultaneous sends and receives, we have the *two-port* model which we have studied in the companion paper [8]. Here we concentrate on the true *one-port* model, where the master cannot be enrolled in more than one communication at any time-step. Although more complicated to analyze, the *one-port* model is more realistic: all the MPI experiments in Section 5 obey this model.

2.2 Scheduling problem

We focus on the following problem: given a time bound T, what is the maximum number of load units that can be processed within time T? As we adopt the classical linear cost model (communication and computation costs are proportional to the number of load units), we can consider that $T = 1$ without loss of generality. Also, owing to the linearity of the model, this problem is equivalent to the problem of minimizing the execution time for a given amount of load to be processed.

A solution to the scheduling problem is characterized by the following information:

- The subset of enrolled processors, and their respective loads. We let α_i denote the amount of load assigned to, and processed by, each participating worker P_i.

- The dates at which each event (incoming communication, computation, return message) starts on each processor.

We can make a few useful simplifications. First, we can assume that each worker starts computing right upon completion of the initial communication from the master. Also, we can assume that the master sends initial messages as soon as possible, i.e. without any delay between two consecutive ones. Symmetrically, we can assume that return messages are sent consecutively, and as late as possible, to the master. However, some idle time may well occur between the time at which a worker P_i has completed its work and the time at which it starts returning the message, just because the master may be busy communicating with another worker. This idle time for P_i will be denoted as x_i. Although it is a classical assumption in DLS theory that all workers work without interruption until the end of the schedule, we point out that we cannot enforce it here *a priori*.

We can now sum up the description of a schedule by:

- a first permutation σ_1 representing the order of sending operations, from the master to each worker,
- a second permutation σ_2 representing the order of the receiving operations, from each worker to the master,
- the quantity α_i of load units sent to each worker P_i,
- the idle time x_i of each worker P_i.

This knowledge allows us to totally describe a schedule, as represented in Figure 2.

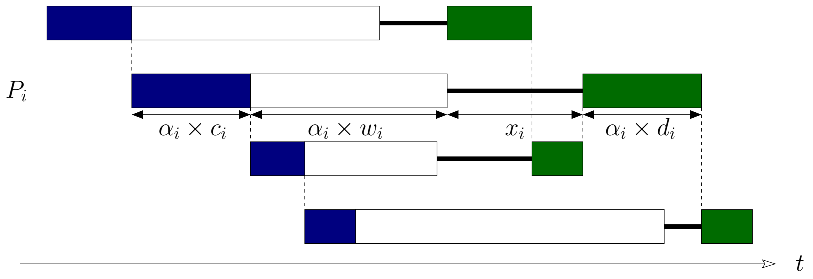
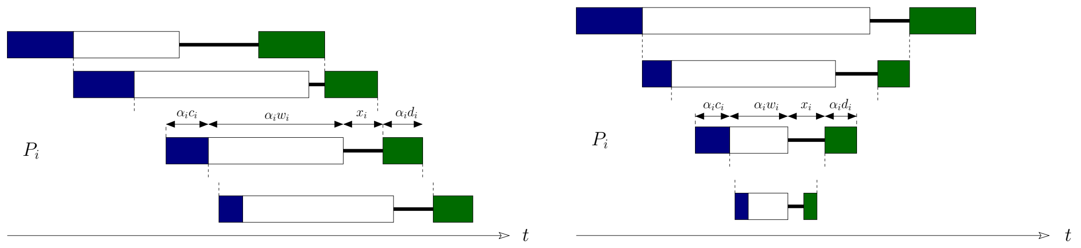


Figure 2: Example of a general schedule. The permutation for input messages is $\sigma_1 = (1, 2, 3, 4)$, while the permutation for output messages is $\sigma_2 = (1, 3, 2, 4)$.

We have not been able to assess a general complexity result, but we have explored special instances of the scheduling problem, namely FIFO schedules for which the order for result messages is the same as the order for input data messages ($\sigma_2 = \sigma_1$): the first worker receiving its data is also the first to send back its results. Figure 3(a) gives an example of such a FIFO schedule.

Note that when the order for results messages is the reverse of the order for input data messages ($\sigma_2 = \sigma_1^R$), we have a LIFO schedule: the first worker receiving its data is the last to send back its results. Figure 3(b) presents an example of such a LIFO schedule. All LIFO schedules naturally obey the one-port model, and we refer to [7, 8] for a characterization of the optimal LIFO schedule.



(a) FIFO

(b) LIFO

Figure 3: Examples of FIFO and LIFO schedules.

2.3 Linear Program for a given scenario

In this section, we show that a linear programming approach can be used to compute the throughput, once the set of participating processors and the ordering of the messages (permutations σ_1 and σ_2) have been determined. We illustrate this approach for a FIFO solution, because we mainly concentrate on FIFO schedules in the following. However, the method can easily be extended to any permutation pair.

Consider the FIFO schedule represented on Figure 3(a), where q processors numbered P_1 to P_q are enrolled in the computation. Consider processor P_i. Before receiving its initial data, P_i has to wait for all previous data transfers, i.e. the time needed for the master to send α_j load units to each processor P_j, for $j < i$; this takes $\sum_{j=1}^{i-1} \alpha_j \times c_j$ time-steps. Then processor P_i receives its data in $\alpha_i \times c_i$ time-steps, and processes it in $\alpha_i \times w_i$ time-steps. Next P_i possibly waits for the communication medium to be free, during x_i times-steps. Finally, processor P_i sends back its results to the master, in $\alpha_i \times d_i$ time-steps. There remains to wait for processors P_{i+1} to P_q to send their results to the master, which requires $\sum_{j=i+1}^{q} \alpha_j \times d_j$ time-steps. Altogether, all this has to be done within a time less than the total execution time $T = 1$, hence we derive the constraint:

$$\sum_{j=1}^{i} \alpha_j \times c_j + \alpha_j \times w_j + \sum_{j=i}^{q} \alpha_j \times d_j + x_i \leq 1 \qquad (1)$$

To enforce one-port constraints, we have to ensure that no communications are overlapped, which translates into:

$$\sum_{i=1}^{q} \alpha_i \times c_i + \sum_{i=1}^{q} \alpha_i \times d_i \leq 1$$

The *throughput* ρ of the schedule the total number of tasks processed in time $T = 1$: $\rho = \sum_{i=1}^{q} \alpha_i$.

To sum it up, given a scenario consisting of a set of q participating processors and two permutations for initial and back communications, we can compute the optimal throughput and derive a schedule achieving such a throughput by solving the following linear program:

$$\textsc{Maximize } \rho = \sum_{i=1}^{q} \alpha_i,\\ \text{UNDER THE CONSTRAINTS}\\ \begin{cases} \text{(2a)} \quad \forall i = 1, \dots q, \quad \sum_{j=1}^{i} \alpha_j \times c_j + \alpha_j \times w_j + \sum_{j=i}^{q} \alpha_j \times d_j + x_i \leq 1 \\ \text{(2b)} \quad \sum_{i=1}^{q} \alpha_i \times c_i + \sum_{i=1}^{q} \alpha_i \times d_i \leq 1 \\ \text{(2c)} \quad \forall i = 1, \dots q, \quad \alpha_i \geq 0 \\ \text{(2d)} \quad \forall i = 1, \dots q, \quad x_i \geq 0 \end{cases} \tag{2}$$

For a given scenario, the cost of this linear programming approach may be acceptable. However, as already pointed out, there is an exponential number of scenarios. Worse, there is an exponential number of FIFO scenarios, even though there is a single permutation to try in this case. The goal of Section 3 is to determine the best FIFO solution in polynomial time.

3 Optimal FIFO schedule on a star platform

In this section, we analyze FIFO schedules. We assume that $d_i = zc_i$ for $1 \leq i \leq p$, with $0 < z < 1$. The case $z > 1$ is symmetrical and will be discussed at the end of the section. The following theorem summarizes the main result:

Theorem 1. *There exists an optimal one-port FIFO schedule where:*

- *processors are ordered in non-decreasing value of c_i;*
- *all processors taking part in the computation have no idle time, except possibly the last one.*

We start the proof with some preliminary lemmas. For a while, we assume that the set of participating processors has been determined, and we come back to this resource selection problem later. The following lemmas provide useful characterizations of an optimal solution:

Lemma 1. *There exists an optimal FIFO one-port schedule where at most one participating processor has some idle time.*

Proof. Let us assume that there are q processors participating in the solution. Any optimal FIFO one-port schedule is an optimal solution to the linear program (2). In this linear program, there are $2q$ unknowns (the α_i's and the x_i's) and $3q + 1$ constraints. Using linear programming theory [25], we know that there is a vertex of the polyhedron defined by the constraints which is optimal. At this optimum, there are $2q$ out of the $3q+1$ constraints which are tight, i.e. which are equalities. Since we are considering q workers participating to the processing, none of the $\alpha_i \geq 0$ constraints is an equality. So, at this optimal vertex, there are $2q$ equalities among the remaining $2q + 1$ constraints: in other words, at most one constraint is not tight. In particular, there exists at most one processor P_i with $x_i > 0$, which means that there is at most one processor with idle time, what achieves the proof of Lemma 1. □

Lemma 2. *There exists an optimal FIFO one-port schedule where only the last participating processor may have some idle time.*

Proof. Let $\mathcal{S}$ be an optimal FIFO one-port schedule, and assume that it involves processors P_1 to P_q in this order. According to Lemma 1, there is at most one processor that may have some idle time between the end of its processing and the start of its backward communication. By contradiction, suppose that this processor is P_i, where $i < q$ and $x_i > 0$. We focus on processors P_i and $P_j = P_{i+1}$. Their activity in $\mathcal{S}$ is represented on Figure 4. We aim at building a schedule $\mathcal{S}'$, where the i-th participating processor has no idle time, but the $i+1$-th participating processor may have idle time, and the number of tasks processed in $\mathcal{S}$' is not smaller than the number of tasks processed in $\mathcal{S}$.

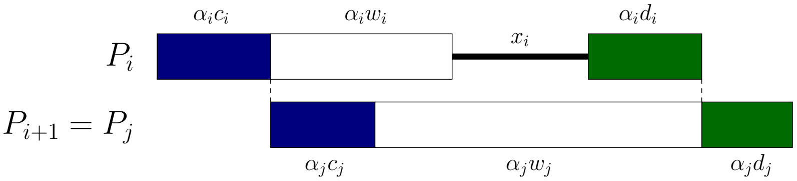


Figure 4: Activity of P_i and P_{i+1} in the initial schedule $\mathcal{S}$

We consider two different cases, according to the communication speeds of P_i and P_j:

- Case 1, $c_i \leq c_j$.

  The transformation used to analyze this case is depicted in Figure 5. Roughly, we keep the overall communication time and the time intervals used for forward and backward communications unchanged, so that other communications (with $P_k \neq P_i, P_j$) are not affected by the transformation. We increase the amount of tasks processed by P_i, and decrease the amount of tasks processed by P_j, so that the overall number of tasks processed by P_i and P_j increases.

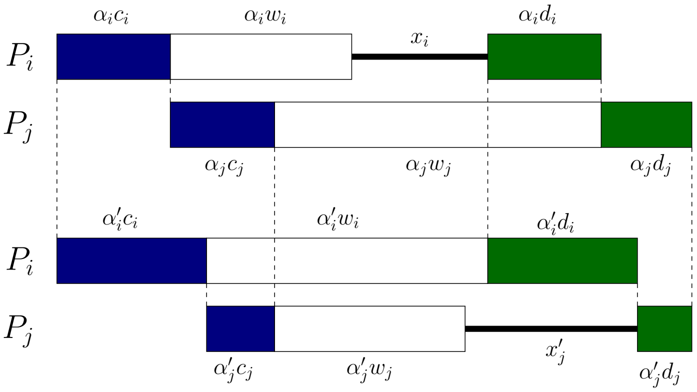


Figure 5: From $\mathcal{S}$ to $\mathcal{S}'$, in the case $c_i \leq c_j$

  More precisely, let us set

$$\alpha'_i = \alpha_i + \frac{x_i}{c_i + w_i}, \qquad \alpha'_j = \alpha_j - \frac{c_i}{c_j}\frac{x_i}{c_i + w_i}, \qquad \alpha'_k = \alpha_k \quad \text{for all } k \neq i, j$$

We set $x'_i = 0$, and we will formulate the x_j value later. The total time needed for data transfers in the new schedule $\mathcal{S}'$ is given by

$$c_i\alpha'_i + c_j\alpha'_j = c_i\alpha_i + c_j\alpha_j + c_i\frac{x_i}{c_i+w_i} - c_j\frac{c_i}{c_j}\frac{x_i}{c_i+w_i} = c_i\alpha_i + c_j\alpha_j$$

Therefore, the total communication time for sending data messages to (P_i, P_j) is the same for $\mathcal{S}$ and $\mathcal{S}'$. In other words, P_j starts computing at the same date in $\mathcal{S}$ and in $\mathcal{S}'$. Since $d_i = zc_i$, we also have

$$d_i\alpha'_i + d_j\alpha'_j = z\left(c_i\alpha'_i + c_j\alpha'_j\right) = z\left(c_i\alpha_i + c_j\alpha_j\right) = d_i\alpha_i + d_j\alpha_j,$$

so that the total time for backward communications is also the same.

The time between the start of the data transfer for P_i and the start of its return transfer is given by

$$\begin{array}{rl} \text{in } \mathcal{S}: & (c_i + w_i)\alpha_i + x_i \\ \text{in } \mathcal{S}': & (c_i+w_i)\alpha'_i = (c_i+w_i)(\alpha_i + \dfrac{x_i}{c_i+w_i}) = (c_i+w_i)\alpha_i + x_i \end{array}$$

So the transfer of the return message for P_i starts at the same date in $\mathcal{S}$ and $\mathcal{S}'$. Together with the fact that communication times are identical in $\mathcal{S}$ and $\mathcal{S}'$ for the group (P_i, P_j), this ensures that we do not perturb the rest of the execution: nothing changes in the data transfers before the date when P_i starts receiving, and after P_j stops receiving. The same holds true for transferring results back to the master.

In $\mathcal{S}'$, P_j has fewer tasks to process than in $\mathcal{S}$, but it can start computing at the same date (since the total data transfer time is the same), and it can stop working later, as its return transfer is shorter. Therefore, we need to introduce some idle time x'_j on P_j. Since the time between the start of its computation and the end of its output transfer is the same for $\mathcal{S}$ and $\mathcal{S}'$, we can write:

$$\begin{array}{rcl} (w_j + d_j)\alpha'_j + x'_j & = & (w_j+d_j)\alpha_j \\ (w_j+d_j) \times \dfrac{-c_i}{c_j}\dfrac{x_i}{c_i+w_i} + x'_j & = & 0 \\ x'_j & = & x_i\left(\dfrac{c_i}{c_j}\dfrac{w_j+d_j}{c_i+w_i}\right) \end{array}$$

Thus, $x'_j > 0$ as soon as $c_i > c_j$, and $x'_j = 0$ otherwise $(c_i = c_j)$. The new schedule $\mathcal{S}'$ is represented in Figure 5, where it is compared to $\mathcal{S}$. The amount of tasks processed by both P_i and P_j in $\mathcal{S}'$ is given by

$$\sum_i \alpha'_i = \sum_i \alpha_i + \epsilon_i - \epsilon_j = \sum_i \alpha_i + \frac{c_j - c_i}{c_j}\frac{x_i}{c_i+w_i}.$$

Since $c_j \geq c_i$, $\mathcal{S}'$ processes at least as many tasks as $\mathcal{S}$ and we have moved the gap one step further from P_i to P_j.

- case 2, $c_i > c_j$.
  In this case, it is not worth moving the gap from P_i to P_j. Therefore, the transformation

(see Figure 6) consists in keeping the gap at P_i, while changing the communication ordering of the FIFO schedule, by switching P_i and P_j. The sketch of the proof is essentially the same as in the first case. We choose the transformation so that the overall communication time for both forward and backward communications to P_i and P_j is the same, thus letting other communications unchanged. Then, we prove that such a transformation increases the overall number of tasks processed by P_i and P_j. More precisely, after the exchange, the number of tasks processed by each processor is the following:

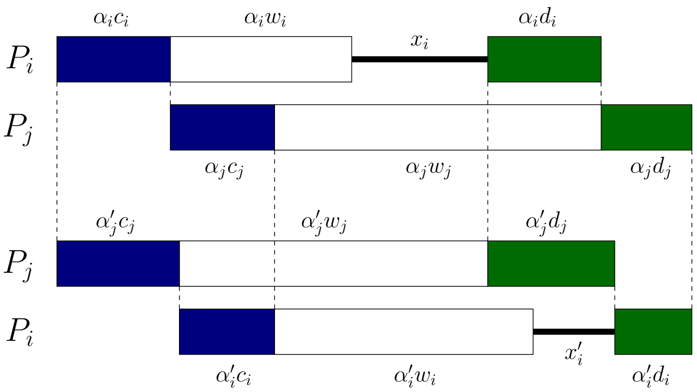


Figure 6: From $\mathcal{S}$ to $\mathcal{S}'$, in the case $c_i > c_j$

$$\begin{aligned}
\alpha'_j &= \alpha_j + \frac{\alpha_i c_i (1-z)}{c_j + w_j} \\
\alpha'_i &= \alpha_i - \frac{\alpha_i c_j (1-z)}{c_j + w_j} \\
\alpha'_k &= \alpha_k \qquad \text{for each processor } P_k \text{ with } k \neq i,j
\end{aligned}$$

Moreover, let us set $x'_k = 0$ for all $k \neq i$ and $x'_i = x_i \left(\frac{w_j + d_j}{c_j + w_j}\right)$. As previously, we need to check that the description of the new schedule in Figure 6 is valid:

- communications for P_i, P_j take the same time in $\mathcal{S}$ and $\mathcal{S}'$:
  The time needed for data transfers for (P_i, P_j) in $\mathcal{S}'$ is

  $$c_j \alpha'_j + c_i \alpha'_i = c_j \alpha_j + c_j \frac{\alpha_i c_i (1-z)}{c_j + w_j} + c_i \alpha_i - c_i \frac{\alpha_i c_j (1-z)}{c_j + w_j} = c_j \alpha_j + c_i \alpha_i$$

  which is the time for data transfers for (P_i, P_j) in $\mathcal{S}$. Since $d_i / c_i = d_j / c_j = z$, the same analysis holds true for output messages.
- P_j starts sending output messages in $\mathcal{S}'$ at the same date as P_i does in $\mathcal{S}$:
  If we assume that the first processor among P_i, P_j starts receiving data at time 0, then P_j starts sending its results at time

  $$T_j = \alpha'_j c_j + \alpha'_j w_j = \left(\alpha_j + \frac{\alpha_i c_i (1-z)}{c_j + w_j}\right) \cdot (c_j + w_j) = \alpha_j (c_j + w_j) + \alpha_i (c_i - d_i)$$

  But in $\mathcal{S}$, we have $\alpha_j (c_j + w_j) = \alpha_i (w_i + d_i) + x_i$, so $T_j = \alpha_i (c_i + w_i) + x_i$, which is exactly the date when the output transfer of P_i starts in $\mathcal{S}$.

- P_i has enough time to process its tasks:
  The time between the beginning of the data transfer for P_i and the beginning of its output transfer in $\mathcal{S}'$ is

$$\begin{aligned} T_i^2 &= \alpha'_i c_i + \alpha'_i w_i + x'_i = \underbrace{\left(\alpha_i - \frac{\alpha_i c_j(1-z)}{c_j+w_j}\right)}_{\alpha_i \frac{w_j+d_j}{c_j+w_j}} (c_i+w_i) + x_i \left(\frac{w_j+d_j}{c_j+w_j}\right) \\ &= \frac{w_j+d_j}{c_j+w_j}\big((c_i+w_i)\alpha_i + x_i\big) \end{aligned}$$

  and the time between the start of P_j's computation and the end of its output transfer in $\mathcal{S}'$ is

$$T_j^2 = \alpha'_j w_j + \alpha'_j d_j = \left(\alpha_j + \frac{\alpha_i(c_i-d_i)}{c_j+w_j}\right)(w_j+d_j).$$

  As previously, we have in $\mathcal{S}$ $\alpha_j(c_j+w_j) = \alpha_i(w_i+d_i)+x_i$, so that we can replace α_j by its actual value in the previous equation:

$$T_j^2 = \left(\frac{\alpha_i(w_i+d_i)+x_i}{c_j+w_j} + \frac{\alpha_i(c_i-d_i))}{c_j+w_j}\right)(w_j+d_j) = \frac{w_j+d_j}{c_j+w_j}\big((c_i+w_i)\alpha_i+x_i\big) = T_i^2.$$

  Therefore, in $\mathcal{S}'$, the end of P_j's output transfer corresponds to the start of P_i's output transfer.

The number of tasks processed in $\mathcal{S}'$ is:

$$\sum_i \alpha'_i = \sum_i \alpha_i + \frac{\alpha_i(c_i-c_j)(1-z)}{c_j+w_j}$$

Since $c_i > c_j$, we have built a FIFO one-port schedule that processes more tasks than $\mathcal{S}$, which is in contradiction with the optimality of $\mathcal{S}$, so the case $c_i > c_j$ never happens.

We apply this approach as many times as needed so that, at the end, the only processor with some idle time is the last enrolled processor P_q. □

We are now able to prove Theorem 1.

Proof. The previous lemmas prove that there exists an optimal FIFO one-port schedule where the only participating processor possibly with idle time is the last one. In order to achieve the proof Theorem 1, we still need to prove that there exists an optimal FIFO schedule where processors are ordered by non-decreasing values of the c_i's.

Let us consider an optimal one-port schedule $\mathcal{S}$ where the last processor only has idle time. We denote by $P_1, \ldots, P_q$ the processors taking part to the computation, in this order. Suppose that processors are not ordered by non-decreasing value of c_i. Then, there exists an index k such that $c_k > c_{k+1}$. We apply the transformation of the second case of the proof of Lemma 2 to processors $P_i = P_k$ and $P_j = P_{k+1}$. Note that this transformation is valid even if there is no idle time for processor P_i $(x_i = 0)$. In this case we get $x'_i = 0$ since $x'_i = x_i\left(\frac{w_j+d_j}{c_j+w_j}\right)$, so that there is no idle time for P_i and P_j in the new schedule $\mathcal{S}'$. However, the total amount of tasks processed in $\mathcal{S}'$ is given by

$$\sum_i \alpha'_i = \sum_i \alpha_i + \frac{\alpha_i(c_i - c_j)(1-z)}{c_j + w_j}.$$

Since $z < 1$ and $c_i > c_j$, $\mathcal{S}'$ processes strictly more tasks than $\mathcal{S}$, which contradicts the optimality of $\mathcal{S}$. Therefore, in $\mathcal{S}$, processors are ordered by non-decreasing values of the c_i's. □

Proposition 1. *An optimal FIFO schedule (with resource selection) can be determined in polynomial time.*

Proof*.* Thanks to Theorem 1, we know that there exists an optimal FIFO schedule where processors are ordered by non-decreasing values of c_i, but we do not know the optimal number of enrolled processors. The following algorithm computes in polynomial time the best throughput, and exhibits an optimal schedule:

1. Sort processors by non-decreasing values of the c_i's: $P_1, \ldots, P_p$.

2. Build a linear program enrolling all p processors, but with an idle-time variable x_i for each of them. This requires the resolution (in rational numbers) of a linear program with $2p$ variables and $3p+1$ constraints.

3. The solution of the linear program provides the set of participating processors (those P_i such that $\alpha_i \neq 0$) and their load.

□

To finish with optimal FIFO schedules on a star, we must deal with the case $z > 1$. We adopt the approach introduced in [8]. Given a FIFO schedule $\mathcal{S}$ on a platform whose parameters are w_i, c_i, d_i and with $z > 1$, we consider the mirror image of $\mathcal{S}$, with time flying backwards, we have a FIFO solution for the platform whose parameters are w_i, d_i, c_i. This simple observation leads to the optimal FIFO solution when $z > 1$: solve the problem with w_i, d_i, c_i as explained in this section (because $c_i = \frac{1}{z} d_i$ with $\frac{1}{z} \leq 1$, Theorem 1 is applicable) and flip over the solution. Note that this implies that initial messages are sent in non-increasing order of the c_i's, rather than in non-decreasing order as was the case for $z < 1$. In passing, this also shows that when $z = 1$, i.e. $c_i = d_i$, the ordering of participating workers has no importance (but some workers may not be enrolled).

4 Optimal FIFO throughput on a bus network

In this section, we give an explicit formula for the optimal throughput of a FIFO schedule, when the platform reduces to a bus network:

Theorem 2. *The optimal FIFO one-port solution when $c_i = c$ and $d_i = d$ achieves the throughput*

$$\rho^{opt} = \min\left\{\frac{1}{c+d}, \frac{\sum_{i=1}^{p} u_i}{1 + d\sum_{i=1}^{p} u_i}\right\}$$

where $u_i = \frac{1}{d+w_i}\prod_{j=1}^{i}\left(\frac{d+w_j}{c+w_j}\right)$. Note that all processors are enrolled in the optimal solution.

Proof. First of all, we show that for a given FIFO one-port schedule the throughput ρ is less than $\min\left\{\frac{1}{c+d}, \frac{\sum_{i=1}^{k} u_i}{1+\sum_{i=1}^{k} u_i d}\right\}$.

Let S be a given one-port schedule enrolling workers P_1 to P_q, and let α_i be the number of load units processed by P_i. As we target a bus network, sending one load unit from the master to any processor takes a time c, whereas receiving one unit from any worker takes a time d. So the time needed to send the total data from the master is: $T_{send} = \sum_{i=1}^{q} \alpha_i c$ and the time needed to receive all results is: $T_{recv} = \sum_{i=1}^{q} \alpha_i d$. As the schedule obeys the one-port model, we know that no reception of the master can start until all sends are completed. Thus we get the constraint $T_{send} + T_{recv} \leq 1$. Hence:

$$T_{send} + T_{recv} = \sum_{i=1}^{q} \alpha_i c + \sum_{i=1}^{q} \alpha_i d = \rho(c+d) \leq 1$$

It remains to show that $\rho \leq \tilde{\rho}$, where $\tilde{\rho} = \frac{\sum_{i=1}^{k} u_i}{1+\sum_{i=1}^{k} u_i d}$. Consider a given FIFO one-port schedule S with throughput ρ. S can be viewed as a two-port schedule, so its throughput cannot exceed that of the optimal throughput in the two-port model. As shown in [7, 8], the optimal two-port solution for a bus network involves all processors and achieves the throughput $\tilde{\rho}$, hence the result.

Then, we prove that their exists a schedule reaching the optimal throughput ρ^{opt}.

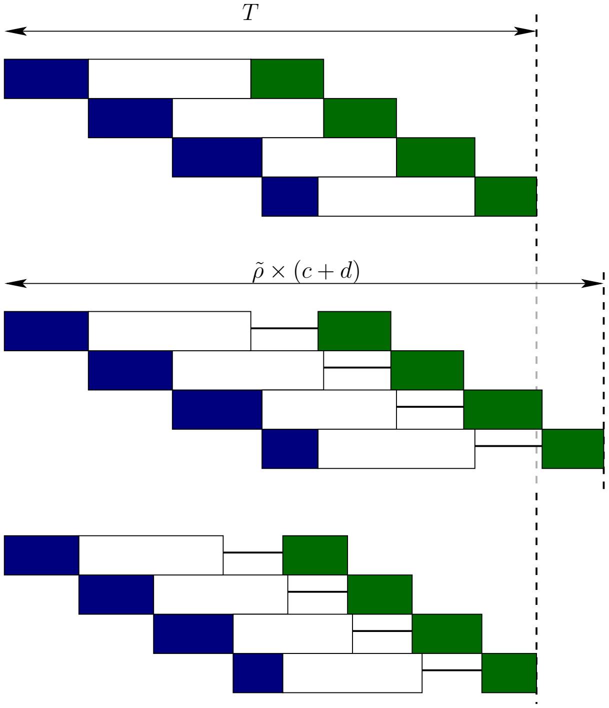


Figure 7: How to transform an optimal two-port schedule into a one-port schedule.

We start with an optimal FIFO two-port schedule S whose throughput is $\tilde{\rho}$ and we transform it into a one-port schedule. We consider two cases:

- $\tilde{\rho} \leq \frac{1}{c+d}$. Then there is no overlap between forth and back communications. The two-port schedule actually is a one-port schedule, whose throughput is $\tilde{\rho} = \rho^{\text{opt}}$.

- $\tilde{\rho} \geq \frac{1}{c+d}$. This is the case with overlap between communication from and to the master, represented in Figure 7. Therefore we have to delay the receptions of the return messages until the master has finished sending data (see second schedule on Figure 7). For this, we add a gap of length $x = \tilde{\rho} \times (c+d) - 1$ between the end of the computation and the transmission of the return message on each processor. The total execution time of the schedule becomes $T' = \tilde{\rho} \times (c+d)$. Finally, to obtain a schedule of total execution time 1, we scale down all quantities by a factor $\frac{1}{\tilde{\rho}\times(c+d)}$ (see the third schedule on Figure 7). We obtain the following characteristics:

  $$\begin{array}{ll} \text{number of tasks processed by } P_i: & \alpha'_i = \dfrac{\alpha_i}{\tilde{\rho} \times (c+d)} \\ \text{gap for every processor:} & x = 1 - \dfrac{1}{\tilde{\rho} \times (c+d)} \end{array}$$

  The throughput of this new schedule is:

  $$\rho' = \frac{\tilde{\rho}}{\tilde{\rho} \times (c+d)} = \frac{1}{c+d}$$

  It remains to show that the new schedule satisfies the conditions for the one-port model. We will show that $T'_{send} + T'_{recv} \leq 1$:

  $$T'_{send} + T'_{recv} = (c+d)\rho' = 1$$

  Hence this schedule obeys the one-port model, and achieves the bound ρ^{opt}. □

5 MPI experiments

In this section we present practical tests using one-port LIFO and FIFO strategies. We choose matrix multiplication as the target application to be implemented on a master/worker platform. The multiplication of two matrices results in a single matrix, hence the initial data message will be twice bigger than the output message: the parameter z introduced in Section 2.1 is equal to 1/2.

Because we deal with a large number M of matrix products to compute, the application can be approximated as a divisible load application, even though side effects are likely to appear: indeed, we need to assign an integer number of matrix products to each worker, instead of a rational number as computed by the linear program.

The objective is to minimize the total execution time for executing the M products. This is a slightly different problem than maximizing the total number of load units processed within a given time bound. However, due to the linear cost model, both problems are equivalent, and the linear programming approach can easily be adapted to this objective.

In our tests we compare the behavior of the following algorithms:

- a FIFO heuristic using all processors, sorted by non-decreasing values of c_i (faster communicating workers first), called INC_C

- a FIFO heuristic using all processors, sorted by non-decreasing values of w_i (faster computing workers first), called INC_W
- the optimal one-port LIFO solution, called LIFO

By construction, the optimal two-port LIFO solution of [7, 8] is indeed a one-port schedule. It involves all processors sorted by by non-decreasing values of c_i. These three heuristics are implemented using the linear programming framework developed in Section 2.3: the choice of the heuristic provides both transfer permutations σ_1 and σ_2; the optimal value of the α_i's, as computed by the linear program, are used for the scheduling. As already mentioned, the solution of the linear program is expressed in rational numbers, but we need to send, process and return integer numbers of matrices. The policy to round the α_i's to integer values is the following. We first round down every value to the immediate lower integer, and then we distribute the K remaining tasks to the first K workers of the schedule in the order of the sending permutation σ_1, by giving one more matrix to process to each of these workers. For instance with 4 processors P_1 to P_4 used in this order for σ_1, if $M = 1000$, $\alpha_1 = 200.4$, $\alpha_2 = 300.2$, $\alpha_3 = 139.8$ and $\alpha_4 = 359.6$, then $K = 2$, and we assign $200 + 1$ matrices to P_1, $300 + 1$ to P_2, 139 to P_3 and 359 to P_4. Obviously, rounding induces some load imbalance, which may slightly impact the actual performance of the heuristics.

5.1 Experimental setting

All tests were made on the cluster *gdsdmi*, which is located within the LIP laboratory at ENS Lyon, and consists in P4 2.4GHz processors with either 256MB or 1GB of memory. In this cluster, 12 nodes are available, so we mainly conduct experiments with one master and 11 workers. To run our test application, we use the *MPICH* implementation [18] of the Message Passing Interface *MPI*. To solve all linear programs, we used the *lp_solve* solver [9]. The total number M of matrices to be processed has been fixed to $M = 1000$.

5.2 Heterogeneity and linear model

The cluster described in the previous section allows us to start with homogeneous conditions: all nodes have the same computation and communicating capabilities. As we target heterogeneous platforms, we simulate heterogeneity by slowing down or speeding up some operation (transfer or computation). We chose to consider the original speed as the slowest available, and sometimes speed up communication or computation when we want to simulate faster workers. For example, to simulate a worker which communicates twice faster than the original speed, we reduce the size of the data and result messages by a factor 2. The data missing for the computation might appear critical, but we are interested in the execution time rather than in the result, so we randomly fill up all the matrices we use. We proceed in the same manner for the computation: simulating a processor which is 5 times faster is done by processing only a fifth of the original computation amount. We have chosen not to slow down transfers or computations in order to avoid problems of memory swapping, which could have happened with very big message sizes, and also in order to reduce execution time.

First, we check that the basic linear cost model which we adopted in our divisible load approach is valid. So we perform a linearity test, by sending different sizes of messages to worker with different (simulated) communication speeds. The results of this test are presented in Figure 8 and show that our assumption on linearity holds true, and that no latency needs to be taken into account.

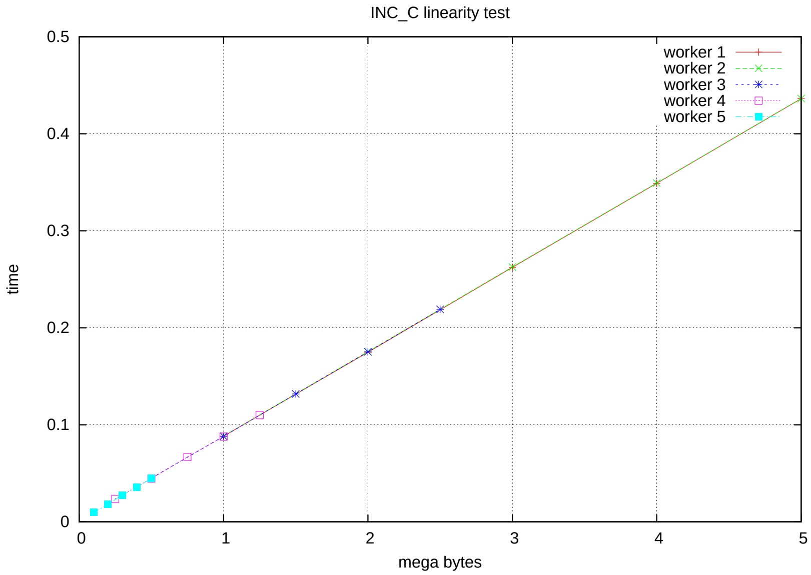


Figure 8: Linearity test with different message sizes, simulating heterogeneous workers.

5.3 Experimental results

5.3.1 Execution analysis

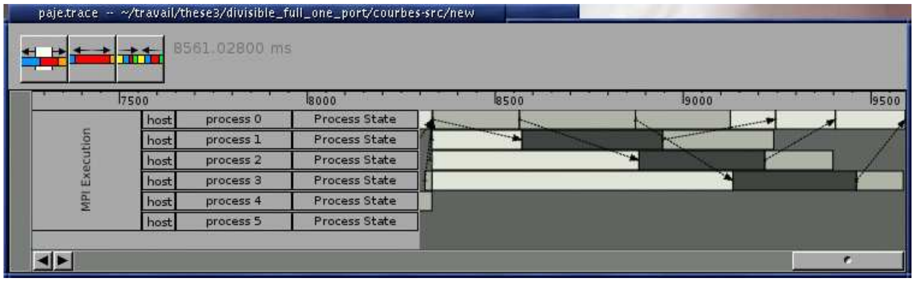


Figure 9: Visualizing an execution on a heterogeneous platform.

To analyze the behavior of the platforms during our experiments, we perform some trace analysis. Figure 9 presents the visualization of one execution. The first line represents the activity of the master, while the other five lines represent workers with different speeds. The initialization of the workers is not shown on this figure. The visualization shows the data transfers (in white), the computation (in dark gray) and the output transfers (in pale gray). Note that for each transfer, the bar "starts" when the receiver is ready for communicating, and "ends" when it has received all data. This explains why at the beginning, all workers start receiving, as they are all waiting for the master to send them some data. As the workers of the platform have (simulated) heterogeneous communicating and computing speeds, not necessary all workers are involved in the computation. Indeed, in the execution shown in

Figure 9, only the first three workers are actually performing some computation. In this experiment, we use FIFO ordering: the sending order is the same for input data and results.

5.3.2 Heuristics comparison

We present here the results of the experiments for a large number of platforms, randomly generated, with parameters varying from 1 to 10, where 1 represents the original speed either for communication or for computation, and 10 represents a worker 10 times faster.

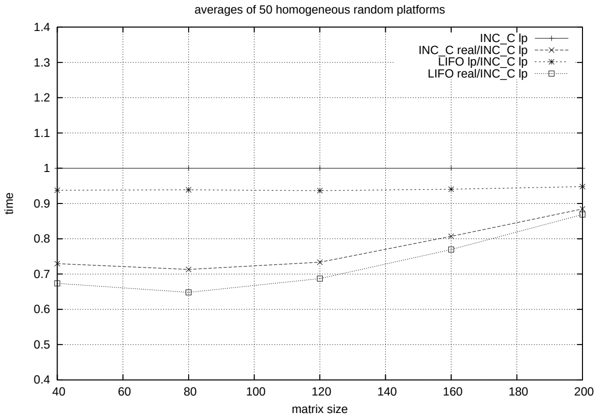


Figure 10: Average of the execution times for 50 homogeneous random platforms, normalized by FIFO theoretical performance.

Homogeneous bus platforms. Figure 10 shows the experimental results for homogeneous bus platforms: in these tests, all workers have the same communication and computation capabilities.

In this figure, as in the following ones, we both plot the actual execution time of each heuristic, after having normalized it against the theoretical prediction for the INC_C heuristic. For example the line "`LIFO real/INC_C lp`" means (execution time of the LIFO heuristic in the experiments)/(theoretical execution time of the INC_C heuristic).

We plot only the INC_C FIFO heuristic because all FIFO strategies are the same with homogeneous communication and computation speeds. In these homogeneous settings, LIFO performs better than FIFO, both in the linear program and in the real experiments.

Heterogeneous bus platforms. Our results on platforms with homogeneous communications and heterogeneous calculation powers are presented on Figure 11. This kind of platform corresponds exactly to the platforms used in Theorem 2. These results supports the theoretical study: INC_C gives better results that INC_W. Again, LIFO performs better than the FIFO strategies. Although the experimental results differ from the theoretical prediction, the theory correctly ranks the different heuristics: in the linear programming approach, LIFO is better than INC_C, which is better than INC_W, and this order in the same in the practical experiments.

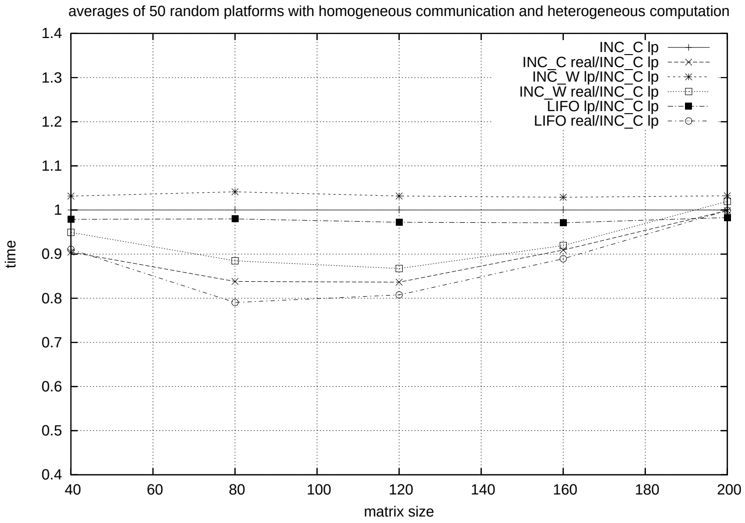


Figure 11: Average of the execution times for 50 random platforms with homogeneous communication power and heterogeneous calculation power, normalized by FIFO theoretical performance.

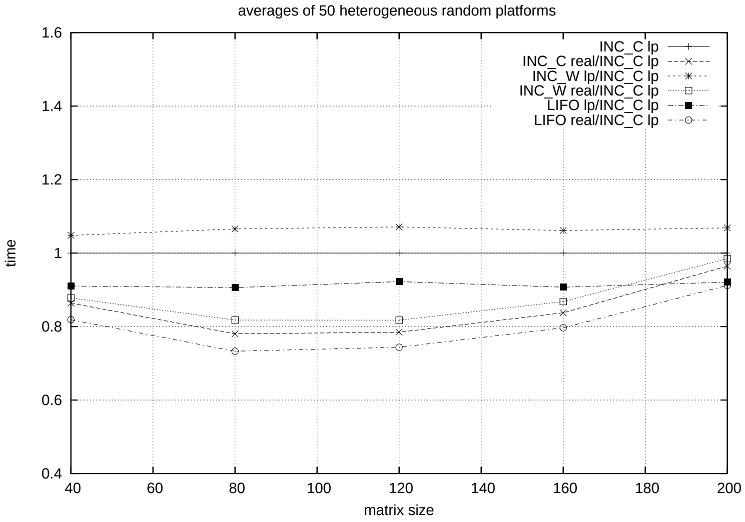
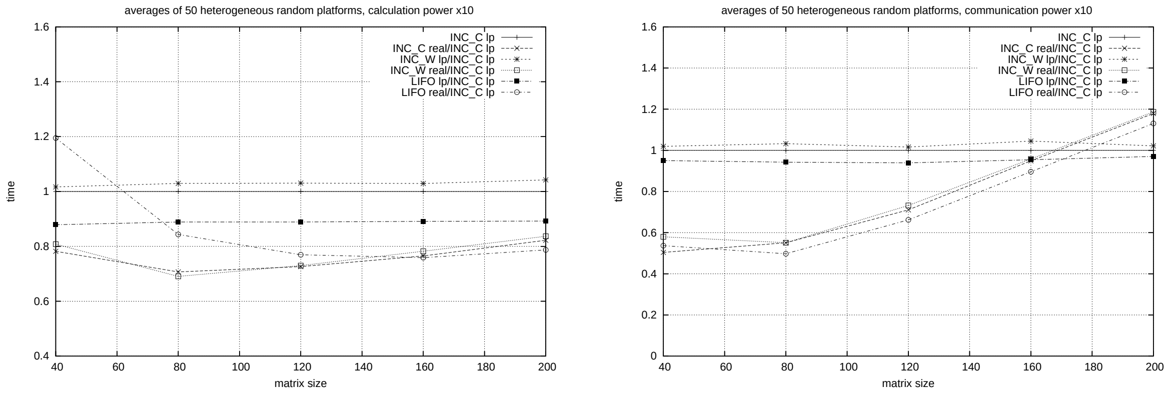


Figure 12: Average of the execution times for 50 heterogeneous random platforms, normalized by FIFO theoretical performance.

Heterogeneous star platforms. Figure 12 presents the results for the case of platforms with heterogeneous communication speed and heterogeneous computation capabilities. Again, INC_C is the best FIFO strategy, as predicted by Theorem 1. The results are very similar to the heterogeneous bus network case: LIFO is better than the FIFO strategies, and the linear program correctly predicts the relative performance of the heuristics, although absolute performance differ from what is predicted by a factor bounded by 20%.

5.3.3 Changing the communication/computation ratio

In this section, we describe the experiments which we performed to better understand the impact of the ratio between communication and computation cost. Starting from the last set of experiments (Figure 12), we first increase the computation power of each processors by a factor 10, and perform the same experiments. Results are presented in Figure 13(a). The test shows that with small matrix sizes, the performances of the LIFO heuristic are much worse than expected, while the performance of both FIFO strategies are very close to each other. This is quite unexpected as the linear program gives a good performance to the LIFO strategy, as in the previous scenarios. The LIFO heuristic might be very sensitive to small performance variations in this case.



(a) Computation 10 times faster

(b) Communication 10 times faster

Figure 13: Average of the execution times for 50 heterogeneous random platforms, normalized by FIFO theoretical performance.

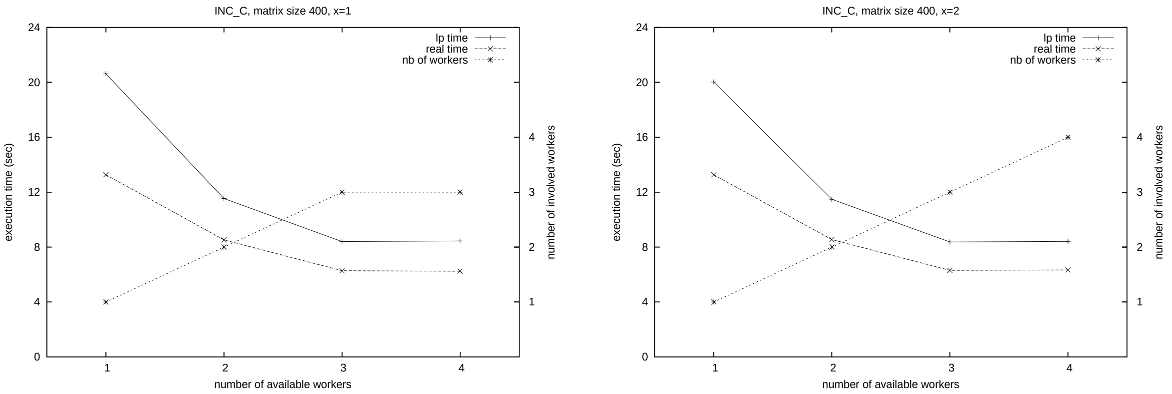
Then, we perform the same experiments with platforms where the communication is 10 times faster than in the original tests, and computation speeds are not changed. Results are presented in Figure 13(b). This shows the limits of the linear cost model, as the ratio between practical performance and theoretical throughput increases linearly with the size of the matrices. However the linear program correctly predicts the relative performance of the different heuristics.

5.3.4 Observing the number of participating workers

As stated earlier, when considering return messages, it can happen that not all workers should be enrolled in the solution to obtain best performances. In this section, we want to check if our framework correctly determines the optimal number of workers that are involved in the computation. We use a platform consisting in 4 workers, where the first 3 workers are fast both in computation and in communication, and the last worker is slower. The following table precisely describes their characteristics:

worker:	1	2	3	4
communication speed:	10	8	8	x
computation speed:	9	9	10	1

Depending on the value of the communication speed x of the last worker, this one should participate or not in the computation. In the following tests, we run our program with a number of slaves from one to four. Then, we record the number of slaves that were really used, and the performance obtained. Figure 14(a) presents the results for $x = 1$. In this case, the last worker is never used (even when we authorize four workers to be used). In Figure 14(b), we present the results for the case $x = 3$. In this case, the fourth worker is used, and the performance is slightly better when using all four workers (even it is hardly noticeable on the graph). This shows that our framework, on this little example, is able to make the right choice about the number of participating processors.



(a) $x = 1$

(b) $x = 3$

Figure 14: Test of participating workers on a heterogeneous platform.

6 Related work

In addition to the landmark book [10], several sources to DLS literature are available: see the two introductory surveys [11, 23], the special issue of the Cluster Computing journal entirely devoted to divisible load scheduling [17], and the Web page collecting DLS-related papers is maintained [22].

DLS applications include linear algebra [13], image processing [19, 21], video and multimedia broadcasting [2, 3], database searching [14, 12], and the processing of large distributed files [26]. These applications are amenable to the simple master-worker programming model and can thus be easily implemented and deployed on computing platforms ranging from small commodity clusters to computational grids [16].

The DLS model comes in two flavors, with a linear cost model and with an affine cost model. The linear cost model is the original model, and has been widely adopted because of its simplicity: several closed-form formulas are available for star, tree, mesh networks among others [10, 22]. The affine cost model (which amounts to introduce a start-up overhead in the communication cost, and/or in the computation cost) has been advocated more recently, for two main reasons: (i) it is more realistic than the linear model; and (ii) it cannot be avoided when dealing with multiple-round scenarios, where the master is allowed to send data to the workers with several messages rather than with a single one. Multiple-round strategies are better amenable to pipelining than one-round approaches, but using a linear cost model would then favor sending a large collection of infinitely small messages, hence the need to add communication latencies. However, latencies render the problem more complex: the DLS problem has recently been proved NP-hard on a star network with the affine model [20].

When dealing with one-round scenarios, as in this paper, the linear model is more realistic, especially when the total work to be distributed to the slaves is large. From a theoretical perspective, one major question was to determine whether adding return messages, while retaining the linear model, would keep the DLS scheduling problem polynomially tractable. We failed to answer this question, but we have been able to characterize optimal solutions for FIFO strategies under the *one-port* model. This result nicely complements our previous study under the *two-port* model.

Relatively few papers have considered adding return messages in the study of DLS problems. Pioneering results are reported by Barlas [4], who tackles the same problem as in this paper (one round, star platform) but with an affine framework model. Barlas [4] concentrates on two particular cases: one called *query processing*, where communication time (both for initial and return messages) is a constant independent of the message size, and the other called *image processing*, which reduces to linear communication times on a bus network, but with affine computation times. In both cases, the optimal sequence of messages is given, and a closed-form solution to the DLS problem is derived. In [15], the authors consider experimental validation of the DLS model for several applications (pattern searching, graph coloring, compression and join operations in databases). They consider both FIFO and LIFO distributions, but they do not discuss communication ordering.

Rosenberg [24] and Adler, Gong and Rosenberg [1] also tackle the DLS model with return messages, but they limit themselves to a bus network (same link bandwidth for all workers). They introduce a very detailed communication model, but they state results for affine communication costs and linear computation costs. They have the additional hypothesis that worker processors can be *slowed down* instead of working at full speed, which allows them to consider no idle times between the end of the execution and the emission of the return messages. They state the very interesting result that all FIFO strategies are equivalent, and that they perform better than any other protocol. Note that our results, although not derived under the same model, are in accordance with these results: when the star platform reduces to a bus platform, the results of Section 3 show that all processors should be involved in the computation, and that their ordering has no impact on the quality of the solution.

Finally, we point out that Altilar and Paker [3] also investigate the DLS problem on a star network, but their paper is devoted to the asymptotic study of several multi-round strategies.

7 Conclusion

In this paper, we present two optimality results for scheduling divisible load on star networks in presence of return messages and under the one-port model. First we are able to characterize the optimal FIFO scheduling on general star networks, where fast-communicating processors should be enrolled first in the computation. We also provide an analytic expression of the total amount of load that can be processed by a FIFO scheduling on homogeneous networks. We have also provided a set of MPI experiments to assess the performance of several heuristics in a real framework.

This paper constitutes the first attempt, to the best of our knowledge, to take return messages into account under the one-port model. Nevertheless, we are still far from the optimal solution of the general problem. Indeed, we are only able to provide optimal schedulings for fixed communication orderings such as FIFO or LIFO. The complexity of finding the pair of optimal permutations for forward and return messages remains open, both under the one-port and two-port models. Despite the simplicity of the linear cost model both for computations and communications, the problem looks very combinatorial, and we conjecture that it is NP-hard.

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] D. Altilar and Y. Paker. An optimal scheduling algorithm for parallel video processing. In *IEEE Int. Conference on Multimedia Computing and Systems*. IEEE Computer Society Press, 1998.

[3] D. Altilar and Y. Paker. Optimal scheduling algorithms for communication constrained parallel processing. In *Euro-Par 2002*, LNCS 2400, pages 197–206. Springer Verlag, 2002.

[4] G.D. Barlas. Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees. *IEEE Trans. Parallel Distributed Systems*, 9(5):429–441, 1998.

[5] S. Bataineh, T.Y. Hsiung, and T.G.Robertazzi. Closed form solutions for bus and tree networks of processors load sharing a divisible job. *IEEE Transactions on Computers*, 43(10):1184–1196, October 1994.

[6] Olivier Beaumont, Henri Casanova, Arnaud Legrand, Yves Robert, and Yang Yang. Scheduling divisible loads on star and tree networks: results and open problems. *IEEE Trans. Parallel Distributed Systems*, 16(3):207–218, 2005.

[7] Olivier Beaumont, Loris Marchal, and Yves Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. In *International Conference on High Performance Computing (HiPC)*, LNCS, Goa, India, 2005. Springer.

[8] Olivier Beaumont, Loris Marchal, and Yves Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. Research Report 2005-21, LIP, ENS Lyon, France, may 2005. Available at www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-21.pdf.

[9] Michel Berkelaar. LP_SOLVE. http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml.

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

[11] 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.

[12] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25:87–98, 1999.

[13] S.K. Chan, V. Bharadwaj, and D. Ghose. Large matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: performance and simulation. *Mathematics and Computers in Simulation*, 58:71–92, 2001.

[14] M. Drozdowski. *Selected problems of scheduling tasks in multiprocessor computing systems*. PhD thesis, Instytut Informatyki Politechnika Poznanska, Poznan, 1997.

[15] Maciej Drozdowski and Pawel Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. In *Proceedings of Euro-Par 2000: Parallel Processing*, LNCS 1900, pages 311–319. Springer, 2000.

[16] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1999.

[17] D. Ghose and T.G. Robertazzi, editors. *Special issue on* Divisible Load Scheduling. Cluster Computing, 6, 1, 2003.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. see also http://www-unix.mcs.anl.gov/mpi/mpich/.

[19] C. Lee and M. Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21:137–160, 1995.

[20] A. Legrand, Y. Yang, and H. Casanova. Np-completeness of the divisible load scheduling problem on heterogeneous star platforms with affine costs. Research Report CS2005-0818, GRAIL Project, University of California at San Diego, march 2005.

[21] X.L. Li, V. Bharadwaj, and C.C. Ko. Distributed image processing on a network of workstations. *Int. J. Computers and Applications (ACTA Press)*, 25(2):1–10, 2003.

[22] T.G. Robertazzi. Divisible Load Scheduling. http://www.ece.sunysb.edu/~tom/dlt.html.

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

[24] 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.

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

[26] R.Y. Wang, A. Krishnamurthy, R.P. Martin, T.E. Anderson, and D.E. Culler. Modeling communication pipeline latency. In *Measurement and Modeling of Computer Systems (SIGMETRICS'98)*, pages 22–32. ACM Press, 1998.

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