Off-line and on-line scheduling on heterogeneous master-slave platforms

Jean-François Pineau, Yves Robert, Frédéric Vivien

Laboratoire de l'Informatique du Parallélisme

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

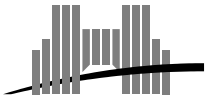
Off-line and on-line scheduling on heterogeneous master-slave platforms

Jean-François Pineau,
Yves Robert,
Frédéric Vivien

July 2005

Research Report N° 2005-31

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



Off-line and on-line scheduling on heterogeneous master-slave platforms

Jean-François Pineau, Yves Robert, Frédéric Vivien

July 2005

Abstract

In this work, we deal with the problem of scheduling independent tasks on heterogeneous master-slave platforms. We target both off-line and on-line problems, with several objective functions (makespan, maximum response time, total completion time). On the theoretical side, our results are two-fold: (i) For off-line scheduling, we prove several optimality results for problems with release dates; (ii) For on-line scheduling, we establish lower bounds on the competitive ratio of any deterministic algorithm. On the practical side, we have implemented several heuristics, some classical and some new ones derived in this paper. We studied experimentally these heuristics on a small but fully heterogeneous MPI platform. Our results show the superiority of those heuristics which fully take into account the relative capacity of the communication links.

Keywords: Scheduling, Master-slave platforms, Heterogeneous computing, On-line, Release dates.

Résumé

Nous nous intéressons ici au problème de l'ordonnancement d'un ensemble de tâches indépendantes sur une plate-forme maître esclave hétérogène. Nous considérons les problèmes en-ligne (ou à la volée) et hors-ligne, pour des fonctions objectives différentes (durée totale d'exécution, temps de réponse maximum, temps de réponse moyen). D'un point de vue théorique, nous obtenons deux types de résultats : (i) pour le problème hors-ligne, nous avons établi plusieurs résultats d'optimalité pour des problèmes avec dates d'arrivée ; (ii) pour le problème en-ligne, nous avons établi des bornes inférieures sur le facteur de compétitivité des algorithmes déterministes. D'un point de vue pratique, nous avons implémenté plusieurs heuristiques, certaines classiques, d'autres issues du présent travail. Nous avons étudié expérimentalement ces heuristiques sur une petite plate-forme MPI totalement hétérogène. Les résultats expérimentaux montrent la supériorité des heuristiques qui prennent complètement en compte les capacités relatives des différents liens de communication.

Mots-clés: Ordonnancement en ligne, Ordonnancement hors-ligne, Calcul hétérogène, Plate-forme maître-esclave

1 Introduction

In this paper, we deal with the problem of scheduling independent tasks on a heterogeneous master-slave platform. We assume that this platform is operated under the *one-port* model, where the master can communicate with a single slave at any time-step. This model is much more realistic than the standard model from the literature, where the number of simultaneous messages involving a processor is not bounded. However, very few complexity results are known for this model (see Section 7 for a short survey). The major objective of this paper is to assess the difficulty of off-line and on-line scheduling problems under the one-port model.

We deal with problems where all tasks have the same size. Otherwise, even the simple problem of scheduling with two identical slaves, without paying any cost for the communications from the master, is NP-hard [12]. Assume that the platform is composed of a master and m slaves $P_1, P_2, \ldots, P_m$. Let c_j be the time needed by the master to send a task to P_j, and let p_j be the time needed by P_j to execute a task. Our main results are the following:

- When the platform is fully homogeneous ($c_j = c$ and $p_j = p$ for all j), we design an algorithm which is optimal for the on-line problem and for three different objective functions (makespan, maximum response time, total completion time).
- When the communications are homogeneous ($c_j = c$ for all j, but different values of p_j), we design an optimal makespan minimization algorithm for the off-line problem with release dates. This algorithm generalizes, and provides a new proof of, a result of Simons [27].
- When the computations are homogeneous ($p_j = p$ for all j, but different value of c_j), we failed to derive an optimal makespan minimization algorithm for the off-line problem with release dates, but we provide an efficient heuristic for this problem.
- For these last two scenarios (homogeneous communications and homogeneous computations), we show that there does not exist any optimal on-line algorithm. This holds true for the previous three objective functions (makespan, maximum response time, total completion time). We even establish lower bounds on the competitive ratio of any deterministic algorithm.

The main contributions of this paper are mostly theoretical. However, on the practical side, we have implemented several heuristics, some classical and some new ones derived in this paper, on a small but fully heterogeneous MPI platform. Our (preliminary) results show the superiority of those heuristics which fully take into account the relative capacity of the communication links.

The rest of the paper is organized as follows. In Section 2, we state some notations for the scheduling problems under consideration. Section 3 deals with fully homogeneous platforms. We study communication-homogeneous platforms in Section 4, and computation-homogeneous platforms in Section 5. We provide an experimental comparison of several scheduling heuristics in Section 6. Section 7 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 8.

2 Framework

To be consistent with the literature [16, 9], we use the notation $\alpha \mid \beta \mid \gamma$ where:

α: the platform– As in the standard, we use P for platforms with identical processors, and Q for platforms with different-speed processors [1]. We add MS to this field to indicate that we work with master-slave platforms.

β: the constraints– We write *on-line* for on-line problems, and r_j when there are release dates. We write $c_j = c$ for communication-homogeneous platforms, and $p_j = p$ for computation-homogeneous platforms.

[1] As we only target sets of same-size tasks, we always fall under the *uniform* processors framework. In other words, the execution time of a task on a processor will only depend on the processor running it and not on the task.

γ: the objective– We let C_i denote the end of the execution of task i. We deal with three objective functions:

- the makespan (total execution time) $max\ C_i$;
- the maximum response time (or maximum flow) $max\ C_i - r_i$: indeed, $C_i - r_i$ is the time spent by task i in the system;
- the total completion time $\sum C_i$, which is equivalent to the sum of the response times $\sum(C_i - r_i)$.

3 Fully homogeneous platforms

For fully homogeneous platforms, we are able to prove the optimality of the *Round-Robin* algorithm which processes the tasks in the order of their arrival, and which assigns them in a cyclic fashion to processors:

Theorem 1. *The* Round-Robin *algorithm is optimal for the problem* $P, MS \mid online, r_j, p_j = p, c_j = c \mid \sum(C_i - r_i)$*, as well as for the minimization of the makespan and of the maximum response time.*

We point out that the complexity of the *Round-Robin* algorithm is linear in the number of tasks and does not depend upon the platform size.

Proof. To prove that the greedy algorithm *Round-Robin* is optimal for our problem, we show that there is an optimal schedule under which the execution of each task starts at the exact same date than under *Round-Robin*. To prove this, we first show two results stating that we can focus on certain particular optimal schedules.

1. There is an optimal schedule such that the master sends the tasks to the slaves in the order of their arrival.

   We prove this result with permutation arguments. Let S be an optimal schedule not verifying the desired property. Remember that the master use its communication links in a sequential fashion. Then we denote by r'_i the date at which the task i arrives on a slave. By hypothesis on S, there are two tasks, j and k, such that j arrives on the master before k, but is sent to a processor slave after k. So:

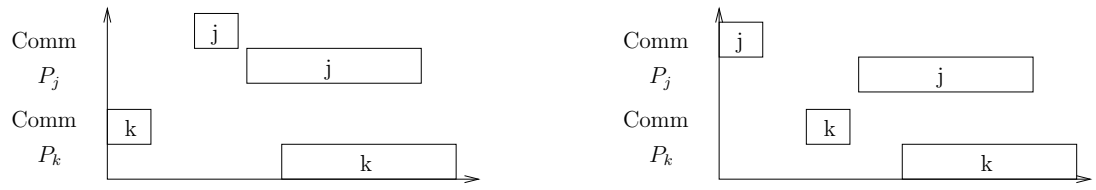
   $$r_j < r_k \text{ and } r'_k < r'_j.$$

   We then define from S a new schedule S' as follows:

   - If the task j was nevertheless treated earlier than the task k (i.e., if $C_j \leq C_k$), then we simply reverse the dispatch dates of tasks j and k, but do not change the processors where they are computed. This is illustrated on Figure 1. In this case, the remainder of the schedule is let unaffected, and the total flow remains the same (just as the makespan, and the maximum flow).
   - If the task j was processed later than the task k, i.e., if $C_j > C_k$, then we send the task j to the processor that was receiving k under S, at the time task k was sent to that processor, and conversely. This is illustrated on Figure 2. Since the tasks j and k have the same size, the use of the processors will be the same, and the remainder of the schedule will remain unchanged. One obtains a new schedule S', having as total flow:

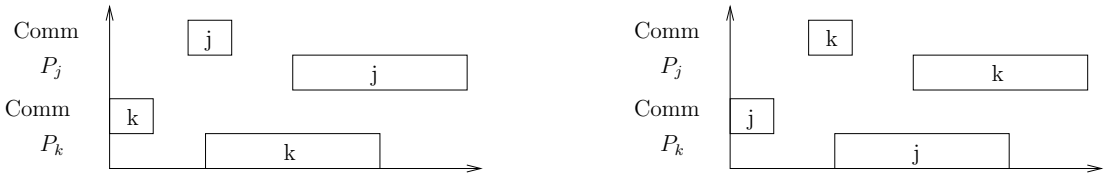
   $$\left(\sum_{\substack{i=1 \\ i \neq j, i \neq k}}^{n} (C_i - r_i) \right) + (C_k - r_j) + (C_j - r_k) = \sum_{i=1}^{n} (C_i - r_i) \tag{1}$$

Therefore, this is also an optimal schedule. In the same way, the makespan as well as the maximum flow are unchanged.



(a) Before permutation (b) After permutation

Figure 1: Permutation on the optimal schedule S (case $C_j \leq C_k$).



(a) Before permutation (b) After permutation

Figure 2: Permutation on the optimal schedule S (case $C_j > C_k$).

By iterating this process, we obtain an optimal schedule where the master sends the tasks according to their arrival dates, i.e., by increasing r_is. Indeed, if one considers the set of the couples $\{(j,k) \mid r_j < r_k \ \& \ r'_j < r'_k\}$, we notice that each iteration of the process strictly increases the size of this set.

2. There is an optimal schedule such that the master sends the tasks to the slaves in the order of their arrival, and such that the tasks are executed in the order of their arrival.

   We will permute tasks to build an optimal schedule satisfying this property from a schedule satisfying the property stated in point 1. Let S be an optimal schedule in which tasks are sent by the master in the order of their arrival. From the above study, we know that such a schedule exists. Let us suppose that S does not satisfy the desired property. Then, there are two tasks j and k, such that

   $$r_j \leq r_k, \quad r'_j < r'_k, \quad \text{and} \quad C_j > C_k.$$

   Then we define a new schedule S' by just exchanging the processors to which the tasks j and k were sent. Then, the task j is computed under S' at the time when k was computed under S, and conversely. This way, we obtain the same total flow $((C_j - r_k) + (C_k - r_j) = (C_j - r_j) + (C_k - r_k))$, the same makespan (since the working times of the processors remains unchanged), whereas the maximum flow can decrease.

Among the optimal schedules which respect the property stated in point 2, we now look at the subset of the solutions computing the first task as soon as possible. Then, among this subset, we

look at the solutions computing the second task as soon as possible. And so on. This way, we define from the set of all optimal schedules an optimal solution, denoted *ASAP*, which processes the tasks in the order of their arrival, and which processes each of them as soon as possible. We will now compare *ASAP* with the schedule *Round-Robin*, formally defined as follows: under *Round-Robin* the task i is sent to the processor $i \ mod \ m$ as soon as possible, while respecting the order of arrival of the tasks.

3. The computation of any task j starts at the same time under the schedules *ASAP* and *Round-Robin*.

   The demonstration is done by induction on the number of tasks. *Round-Robin* sends the first task as soon as possible, just as *ASAP* does. Let us suppose now that the first j tasks satisfy the property. Let us look at the behavior of *Round-Robin* on the arrival of the $(j+1)$-th task. The computation of the $(j+1)$-th task starts at time:

   $$RR(j+1) = max\left\{r'_{j+1}, RR(j+1-m)+p\right\}.$$

   Indeed, either the processor is available at the time the task arrives on the slave, and the task execution starts as soon as the task arrives, i.e., at time r'_{j+1}, or the processor is busy when the task arrives. In the latter case, the processor will be available when the last task it previously received (i.e., the $(j+1-m)$-th task according to the *Round-Robin* strategy) will be completed, at time $RR(j+1-m)+p$.

   Therefore, if $RR(j+1) = r'_{j+1}$, *Round-Robin* remains optimal, since the task is processed as soon as it is available on a slave, and since it was sent as soon as possible. Otherwise, $RR(j+1) = RR(j+1-m)+p$. But, by induction hypothesis, we know that $\forall \lambda, 1 \leq \lambda \leq m, RR(j+1-\lambda) = ASAP(j+1-\lambda)$. Furthermore, thanks to the *Round-Robin* scheduling policy, we know that $\forall i, RR(i) \leq RR(i+1)$. Therefore:

   $$\forall \lambda, 1 \leq \lambda \leq m, RR(j+1-m) \leq RR(j+1-\lambda) < RR(j+1-m)+p = RR(j+1)$$

   This implies that, between $RR(j+1-m)$ and $RR(j)$, m tasks of size p were started, under *Round-Robin*, and also under *ASAP* because of the induction hypothesis. Therefore, during that time interval, m slaves were selected. Then, until the date $RR(j+1-m)+p$, all the slaves are used and, thus, the task $j+1$ is launched as soon as possible by *Round-Robin*, knowing that *ASAP* could not have launched it earlier. Therefore, $ASAP(j+1) = RR(j+1)$. We can conclude.

We have already stated that the demonstrations of points 1 and 2 are valid for schedules minimizing either makespan, total flow, or maximum flow. The reasoning followed in the demonstration of point 3 is independent from the objective function. Therefore, we demonstrated the optimality of *Round-Robin* for these three objective functions. □

4 Communication-homogeneous platforms

In this section, we have $c_j = c$ but different-speed processors. We order them so that P_1 is the fastest processor (p_1 is the smallest computing time p_i), while P_m is the slowest processor.

4.1 On-line scheduling

Theorem 2. *There is no scheduling algorithm for the problem* $P, MS \mid online, r_i, p_j, c_j = c \mid max\ C_i$ *with a competitive ratio less than* $\frac{5+3\sqrt{5}}{10}$.

Proof. Suppose the existence of an on-line algorithm $\mathcal{A}$ with a competitive ratio $\rho = \frac{5+3\sqrt{5}}{10} - \epsilon$, with $\epsilon > 0$. We will build a platform and study the behavior of $\mathcal{A}$ opposed to our adversary. The platform consists of two processors, where $p_1 = 2$, $p_2 = \frac{1+3\sqrt{5}}{2}$, and $c = 1$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ sends the task i either on P_1, achieving a makespan at least equal to 3, or on P_2, with a makespan at least equal to $\frac{3+3\sqrt{5}}{2}$. At time-step 1, we check if $\mathcal{A}$ made a decision concerning the scheduling of i, and the adversary reacts consequently:

1. If $\mathcal{A}$ did not begin the sending of the task i, the adversary does not send other tasks. The best makespan is then 4. As the optimal makespan is 3, we have a competitive ratio of $\frac{4}{3} > \frac{5+3\sqrt{5}}{10}$. This refutes the assumption on ρ. Thus the algorithm $\mathcal{A}$ must have scheduled the task i at time 1.

2. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible makespan is then equal to $\frac{3+3\sqrt{5}}{2}$, which is even worse than the previous case. Consequently, algorithm $\mathcal{A}$ does not have another choice than to schedule the task i on P_1.

At time-step 1, the adversary sends another task, j. In this case, we look, at time-step 2, at the assignment $\mathcal{A}$ made for j:

1. If j is sent on P_2, the adversary does not send any more task. The best achievable makespan is then $\frac{5+3\sqrt{5}}{2}$, whereas the optimal is 5. The competitive ratio is then $\frac{5+3\sqrt{5}}{10} > \rho$.

2. If j is sent on P_1 the adversary sends a last task at time-step 2. The best possible makespan is then $\frac{7+3\sqrt{5}}{2}$, whereas the optimal is $\frac{5+3\sqrt{5}}{2}$. The competitive ratio is still $\frac{5+3\sqrt{5}}{10}$, higher than ρ.

□

Remark 1. *Similarly, we can show that there is no on-line scheduling for the problem* $P, MS \mid online, r_i, p_j, c_j = c \mid \sum C_i$ *whose competitive ratio* ρ *is strictly lower than* $\frac{2+4\sqrt{2}}{7}$*, and that there is no on-line scheduling for the problem* $P, MS \mid online, r_i, p_j, c_j = c \mid max\ (C_i - r_i)$ *whose competitive ratio* ρ *is strictly lower than* $\frac{7}{6}$*.*

4.2 Off-line scheduling

In this section, we aim at designing an optimal algorithm for the off-line version of the problem, with release dates. We target the objective max C_i. Intuitively, to minimize the completion date of the task arriving last, it is necessary to allocate this task to the fastest processor (which will finish it the most rapidly). However, the other tasks should also be assigned so that this fastest processor will be available as soon as possible for the task arriving last. We define the greedy algorithm *SLJF (Scheduling Last Jobs First)* as follows:

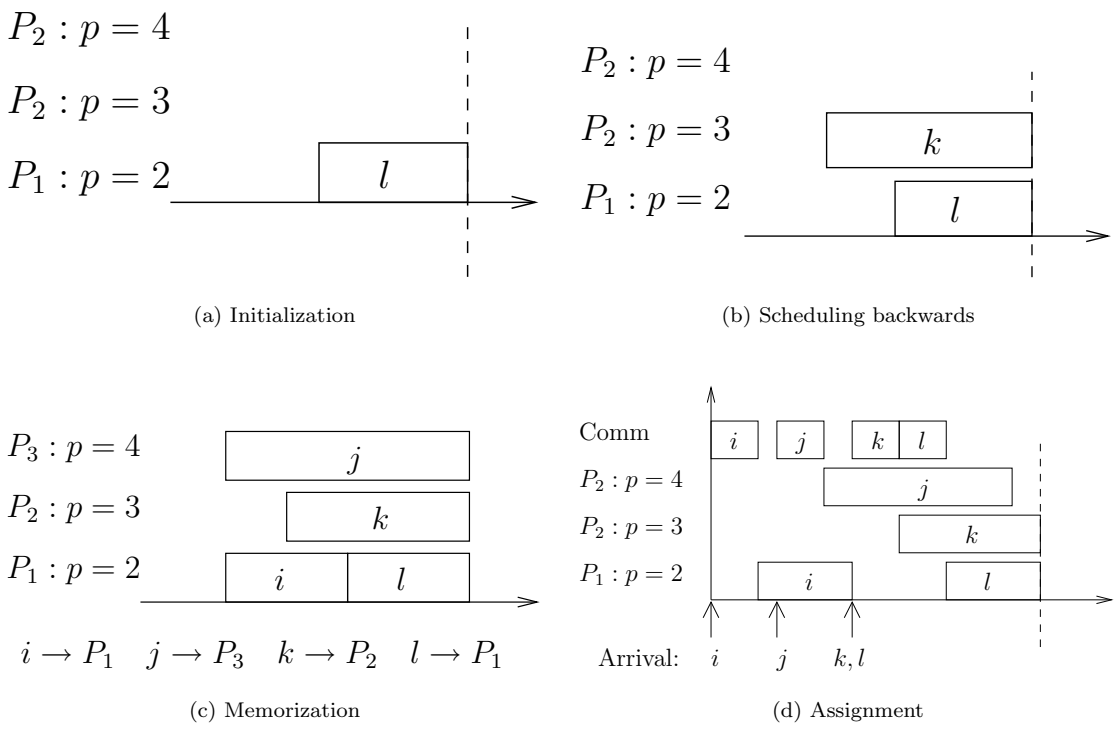
Initialization– Take the last task which arrives in the system and allocate it to the fastest processor (Figure 3(a)).

Scheduling backwards– Among the not-yet-allocated tasks, select the one which arrived latest in the system. Assign it, without taking its arrival date into account, to the processor which will begin its execution at the latest, but without exceeding the completion date of the previously scheduled task (Figure 3(b)).

Memorization– Once all tasks are allocated, record the assignment of the tasks to the processors (Figure 3(c)).

Assignment– The master sends the tasks according to their arrival dates, as soon as possible, to the processors which they have been assigned to in the previous step (Figure 3(d)).

Theorem 3. SLJF *is an optimal algorithm for the problem* $Q, MS \mid r_j, p_j, c_j = c \mid max\ C_i$*.*



(a) Initialization (b) Scheduling backwards

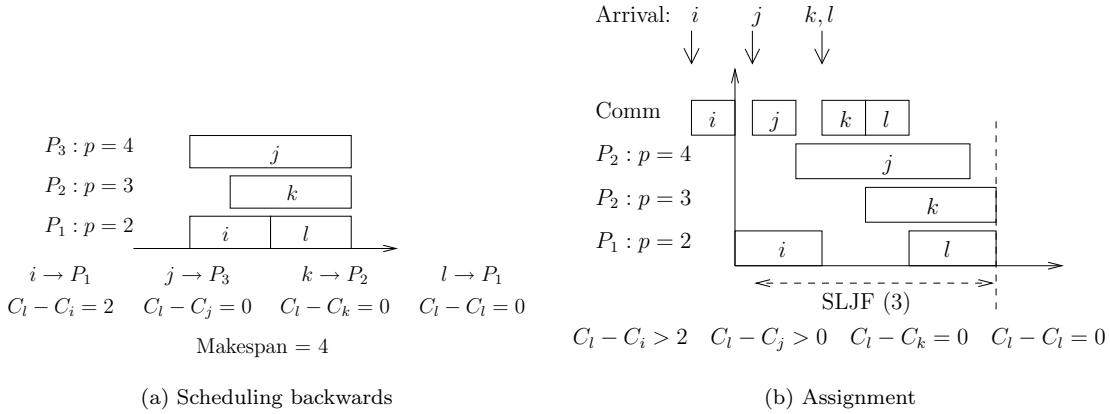
(c) Memorization (d) Assignment

Figure 3: Different steps of the SLJF algorithm, with four tasks i, j, k, and l.

Proof. The first three phases of the SLJF algorithm are independent of the release dates, and only depend on the number of tasks which will arrive in the system. The proof proceeds in three steps. First we study the problem without communication costs, nor release dates. Next, we take release dates into account. Finally, we extend the result to the case with communications. The second step is the most difficult.

For the first step, we have to minimize the makespan during the scheduling of identical tasks with heterogeneous processors, without release dates. Without communication costs, this is a well-known load balancing, problem, which can be solved by a greedy algorithm [6]. The "scheduling backwards" phase of *SLJF* solves this load balancing problem optimally. Since the problem is without release dates, the "memorization" phase does not increase the makespan, which thus remains optimal.

Next we add the constraints of release dates. To show that SLJF is optimal, we proceed by induction on the number of tasks. For a single task, it is obvious that the addition of a release date does not change anything about the optimality of the solution. Let us suppose the algorithm optimal for n tasks, or less. Then look at the behavior of the algorithm to process $n+1$ tasks. If the addition of the release dates does not increase the makespan compared to that obtained during the "memorization" step, then an optimal scheduling is obtained. If not, let us look once again at the problem starting from the end. Compare the completion times of the tasks in the scheduling of the "memorization" phase (denoted as $(C_n - C_i)_{\text{memo}}$), and in the "assignment" phase (denoted as $(C_n - C_i)_{\text{final}}$). If both makespans are equal, we are finished. Otherwise, there are tasks such that $(C_n - C_i)_{\text{memo}} < (C_n - C_i)_{\text{final}}$. Let j be the last task satisfying this property. In this case, the scheduling of the $(n-j-1)$ last tasks corresponds to *SLJF* in the case of $(n-j-1)$ tasks, when the first task arrives at time r_{j+1} (see Figure 4). And since j is the last task satisfying the above property, we are sure that the processors are free at the expected times. Using the induction hypothesis, scheduling is thus optimal from r_{j+1}, and task $j+1$ cannot begin its computation



(a) Scheduling backwards (b) Assignment

Figure 4: Detailing the last two phases of the SLJF algorithm.

earlier. The whole scheduling is thus optimal. Finally, *SLJF* is optimal to minimize the makespan in the presence of release dates.

Taking communications into account is now easy. Under the one-port model, with a uniform communication time for all tasks and processors, the optimal policy of the master consists in sending the tasks as soon as they arrive. Now, we can consider the dates at which the tasks are available on the slaves, and consider them as *release dates* for a problem without communications. □

Remark 2. *It should be stressed that, by posing $c = 0$, our approach allows to provide a new proof to the result of Barbara Simons [27].*

5 Computation-homogeneous platforms

In this section, we have $p_j = p$ but processor links with different capacities. We order them, so that P_1 is the fastest communicating processor (c_1 is the smallest computing time c_i).

5.1 On-line scheduling

Just as in Section 4, we can bound the competitive ratio of any deterministic algorithm:

Theorem 4. *There is no scheduling algorithm for the problem $P, MS \mid online, r_i, p_j = p, c_j \mid \max C_i$ whose competitive ratio ρ is strictly lower than $\frac{6}{5}$.*

Proof. Assume that there exists a deterministic on-line algorithm $\mathcal{A}$ whose competitive ratio is $\rho = \frac{6}{5} - \epsilon$, with $\epsilon > 0$. We will build a platform and an adversary to derive a contradiction. The platform is made up with two processors P_1 and P_2 such that $p_1 = p_2 = p = \max\{5, \frac{12}{25\epsilon}\}$, $c_1 = 1$ and $c_2 = \frac{p}{2}$.

Initially, the adversary sends a single task i at time 0. $\mathcal{A}$ executes the task i, either on P_1 with a makespan at least[2] equal to $1 + p$, or on P_2 with a makespan at least equal to $\frac{3p}{2}$.

At time-step $\frac{p}{2}$, we check whether $\mathcal{A}$ made a decision concerning the scheduling of i, and which one:

1. If $\mathcal{A}$ scheduled the task i on P_2 the adversary does not send other tasks. The best possible makespan is then $\frac{3p}{2}$. The optimal scheduling being of makespan $1+p$, we have a competitive

[2] Nothing forces $\mathcal{A}$ to send the task i as soon as possible.

ratio of

$$\rho \geq \frac{\frac{3p}{2}}{1+p} = \frac{3}{2} - \frac{3}{2(p+1)} > \frac{6}{5}$$

because $p \geq 5$ by assumption. This contradicts the hypothesis on ρ. Thus the algorithm $\mathcal{A}$ cannot schedule task i on P_2.

2. If $\mathcal{A}$ did not begin to send the task i, the adversary does not send other tasks. The best makespan that can be achieved is then equal to $\frac{p}{2} + (1+p) = 1 + \frac{3p}{2}$, which is even worse than the previous case. Consequently, the algorithm $\mathcal{A}$ does not have any other choice than to schedule task i on P_1.

At time-step $\frac{p}{2}$, the adversary sends three tasks, j, k and l. No schedule which executes three of the four tasks on the same processor can have a makespan lower than $1+3p$ (minimum duration of a communication and execution without delay of the three tasks). We now consider the schedules which compute two tasks on each processor. Since i is computed on P_1, we have three cases to study, depending upon which other task (j, k, or l) is computed on P_1:

1. If j is computed on P_1:

   (a) Task i is sent to P_1 during the interval $[0, 1]$ and is computed during the interval $[1, 1+p]$.

   (b) Task j is sent to P_1 during the interval $[\frac{p}{2}, 1 + \frac{p}{2}]$ and is computed during the interval $[1+p, 1+2p]$.

   (c) Task k is sent to P_2 during the interval $[1+\frac{p}{2}, 1+p]$ and is computed during the interval $[1+p, 1+2p]$.

   (d) Task l is sent to P_2 during the interval $[1+p, 1+\frac{3p}{2}]$ and is computed during the interval $[1+2p, 1+3p]$.

   The makespan of this schedule is then $1+3p$.

2. If k is computed on P_1:

   (a) Task i is sent to P_1 during the interval $[0, 1]$ and is computed during the interval $[1, 1+p]$.

   (b) Task j is sent to P_2 during the interval $[\frac{p}{2}, p]$ and is computed during the interval $[p, 2p]$.

   (c) Task k is sent to P_1 during the interval $[p, 1+p]$ and is computed during the interval $[1+p, 1+2p]$.

   (d) Task l is sent to P_2 during the interval $[1+p, 1+\frac{3p}{2}]$ and is computed during the interval $[2p, 3p]$.

   The makespan of this scheduling is then $3p$.

3. If l is computed on P_1:

   (a) Task i is sent to P_1 during the interval $[0, 1]$ and is computed during the interval $[1, 1+p]$.

   (b) task j is sent to P_2 during the interval $[\frac{p}{2}, p]$ and is computed during the interval $[p, 2p]$.

   (c) Task k is sent to P_2 during the interval $[p, \frac{3p}{2}]$ and is computed during the interval $[2p, 3p]$.

   (d) Task l is sent to P_1 during the interval $[\frac{3p}{2}, 1+\frac{3p}{2}]$ and is computed during the interval $[1+\frac{3p}{2}, 1+\frac{5p}{2}]$.

   The makespan of this schedule is then $3p$.

Consequently, the last two schedules are equivalent and are better than the first. Altogether, the best achievable makespan is $3p$. But a better schedule is obtained when computing i on P_2, then j on P_1, then k on P_2, and finally l on P_1. The makespan of the latter schedule is equal to $1 + \frac{5p}{2}$. The competitive ratio of algorithm $\mathcal{A}$ is necessarily larger than the ratio of the best reachable makespan (namely $3p$) and the optimal makespan, which is not larger than $1 + \frac{5p}{2}$. Consequently:

$$\rho \geq \frac{3p}{1+\frac{5p}{2}} = \frac{6}{5} - \frac{6}{5(5p+2)} > \frac{6}{5} - \frac{6}{25p} \geq \frac{6}{5} - \frac{\epsilon}{2}$$

which contradicts the assumption $\rho = \frac{6}{5} - \epsilon$ with $\epsilon > 0$. □

5.2 Off-line scheduling

In the easy case where $\sum_{i=1}^{p} c_i \leq p$, and without release dates, *Round-Robin* is optimal for makespan minimization. But in the general case, not all slaves will be enrolled in the computation. Intuitively, the idea is to use the fastest m' links, where m' is computed so that the time p to execute a task lies between the time necessary to send a task on each of the fastest $m'-1$ links and the time necessary to send a task on each of the fastest m' links. Formally,

$$\sum_{i=1}^{m'-1} c_i < p \quad \text{and} \quad \sum_{i=1}^{m'} c_i \geq p.$$

With only m' links selected in the platform, we aim at deriving an algorithm similar to *Round-Robin*. But we did not succeed in proving the optimality of our approach. Hence the algorithm below should rather be seen as a heuristic.

The difficulty lies in deciding when to use the m'-th processor. In addition to be the one having the slowest communication link, its use can cause a moment of inactivity on another processor, since $\sum_{i=1}^{m'-1} c_i + c_{m'} \geq p$. Our greedy algorithm will simply compare the performances of two strategies, the one sending tasks only on the $m'-1$ first processors, and the other using the m'-th processor "at the best possible moment".

Let *RRA* be the algorithm sending the tasks to the $m'-1$ fastest processors in a cyclic way, starting with the fastest processor, and scheduling the tasks in the reverse order, from the last one to the first one. Let *RRB* be the algorithm sending the last task to processor m', then following the *RRA* policy. We see that *RRA* seeks to continuously use the processors, even though idle time may occur on the communication link, and on the processor $P_{m'}$. On the contrary, *RRB* tries to continuously use the communication link, despite leaving some processors idle.

The global behavior of the greedy algorithm, *SLJFWC (Scheduling the Last Job First With Communication)* is as follows:

Initialization: Allocate the $m'-1$ last tasks to the fastest $m'-1$ processors, from the fastest to the slowest.

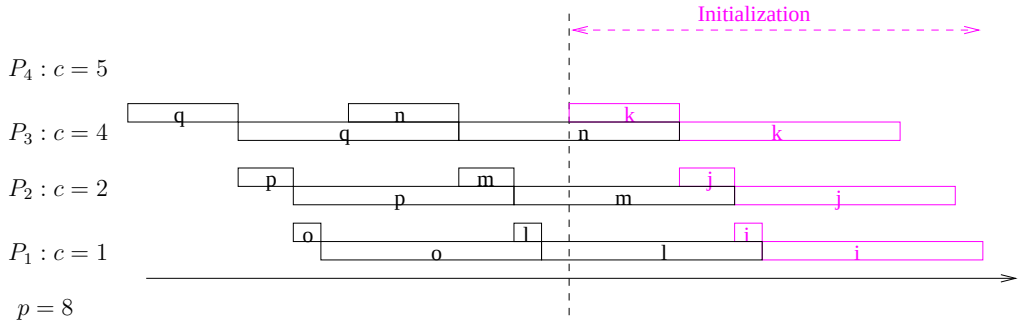
Comparison: Compare the schedules *RRA* and *RRB*. If there are not enough tasks to enforce the following *stop and save* condition, then keep the fastest policy (see Figure 5).

Stop and save: After $k(m'-1)+1$ allocated tasks ($k \geq 2$), if (see Figure 6)

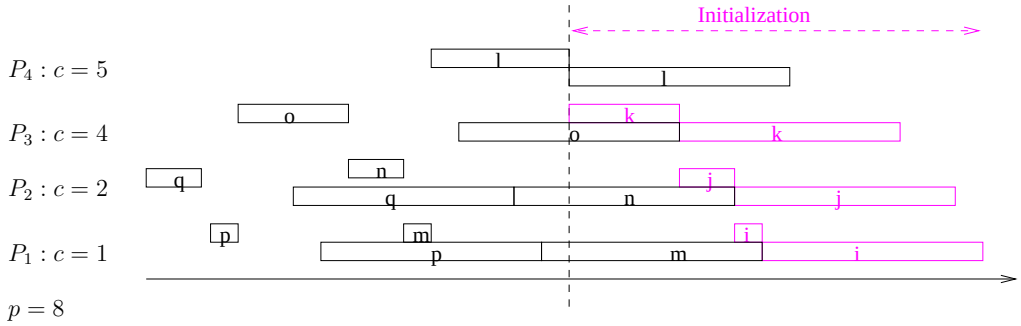
$$\begin{cases} k\sum_{i=1}^{m'-1} c_i + c_{m'} > kp \\ (k+1)\sum_{i=1}^{m'-1} c_i + c_{m'} \leq (k+1)p \end{cases}$$

then keep the task assignment of *RRB* for the last $k(m'-1)+1$ tasks, and start again the comparison phase for the remaining tasks. If not, proceed with the comparison step.

End: When the last task is treated, keep the fastest policy.



(a) Algorithm *RRA*



(b) Algorithm *RRB*

Figure 5: Algorithms *RRA* and *RRB* with 9 tasks.

The intuition under this algorithm is simple. We know that if we only have the $m' - 1$ fastest processors, then *RRA* is optimal to minimize the makespan. However, the time necessary for sending a task on each of the $m' - 1$ processors is lower than p. This means that the sending of the tasks takes "advances" compared to their execution. This advance, which accumulates for all the $m' - 1$ tasks, can become sufficiently large to allow the sending of a task on another m-th processor, for "free", i.e. without delaying the treatment of the next tasks to come on the other processors.

6 MPI experiments

6.1 The experimental platform

We build a small heterogeneous master-slave platform with five different computers, connected to each other by a fast Ethernet switch (100 Mbit/s). The five machines are all different, both in terms of CPU speed and in the amount of available memory. The heterogeneity of the communication links is mainly due to the differences between the network cards. Each task will be a matrix, and each slave will have to calculate the determinant of the matrices that it will receive. Whenever needed, we play with matrix sizes so as to achieve more heterogeneity in the CPU speeds or communication bandwidths.

Below we report experiments for the following configuration (in an arbitrary unit):

- $c_1 = 0.011423$ et $p_1 = 0.052190$
- $c_2 = 0.012052$ et $p_2 = 0.019685$

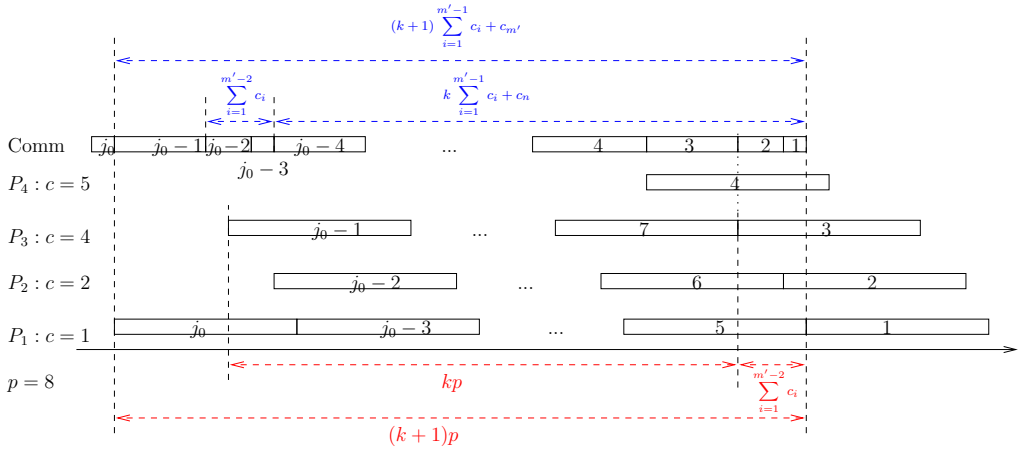


Figure 6: The *stop and save* condition.

- $c_3 = 0.016808$ et $p_3 = 0.101777$
- $c_4 = 0.043482$ et $p_4 = 0.288397$

6.2 Results

Figure 7 shows the makespan obtained with classical scheduling algorithms, such as *SRPT (Shortest Remaining Processing Time)*, *List Scheduling*, and several variants of *Round-Robin*, as well as with $SLJF$ and $SLJFWC$. In this experinebt, all the tasks to be scheduled arrived at time 0 (off-line framework without release dates).

Each point on the figure, representing the makespan of a schedule, corresponds in reality to an average obtained while launching several times the experiment. We see that $SLJFWC$ obtains good results. $SLJF$ remains competitive, even if it was not designed for a platform with different communications links.

Figure 8 also represents the average makespan of various algorithms, but on a different platform. This time, the parameters were modified by software in order to render the processors homogeneous. In this case, $SLJFWC$ is still better, and $SLJF$ obtains poor performances.

Finally, Figure 9 represents the average makespan in the presence of release-dates. Again, $SLJFWC$ performs well, even though it was not designed for problems with release-dates.

7 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 [7, 8] 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 [24], who report

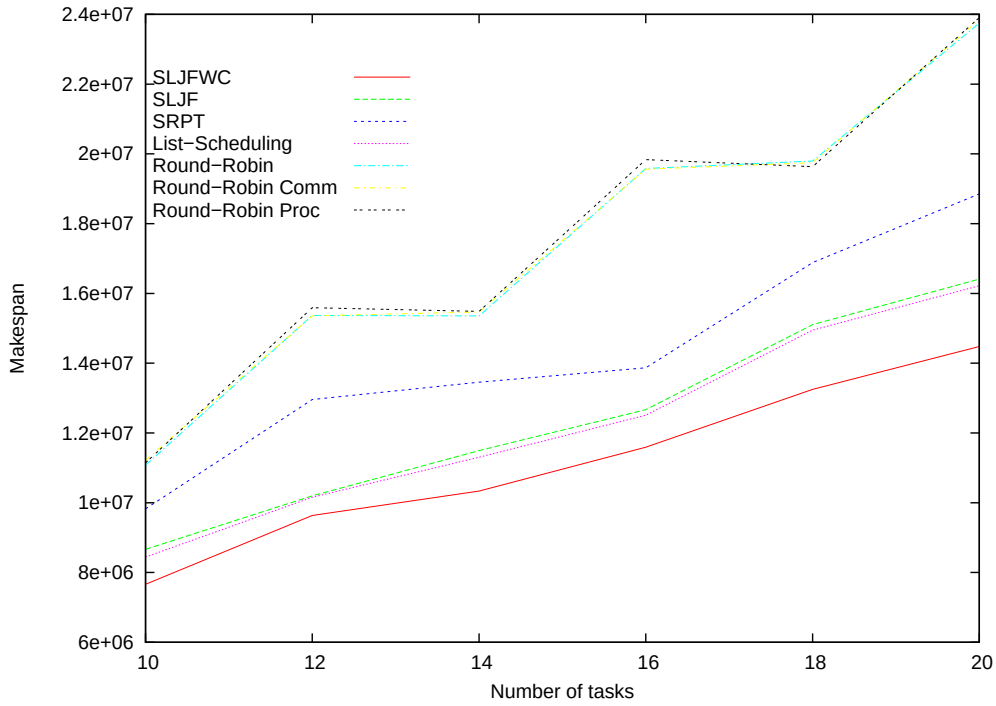


Figure 7: Comparing the makespan of several algorithms.

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. [1], Liu [19], and Khuller and Kim [15]. 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 [2, 3] depart from the one-port model as they allow a sending processor to initiate another communication while a previous one is still on-going 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.

Task graph scheduling– Task graph scheduling is usually studied using the so-called *macro-dataflow* model [20, 26, 10, 11], whose major flaw is that communication resources are not limited. In this model, a processor can send (or receive) any number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Also, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously occur on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units. More recent papers [29, 21, 23, 4, 5, 28] take communication resources into account.

Hollermann et al. [13] and Hsu et al. [14] introduce the following model for task graph scheduling: each processor can either send or receive a message at a given time-step (bidi-

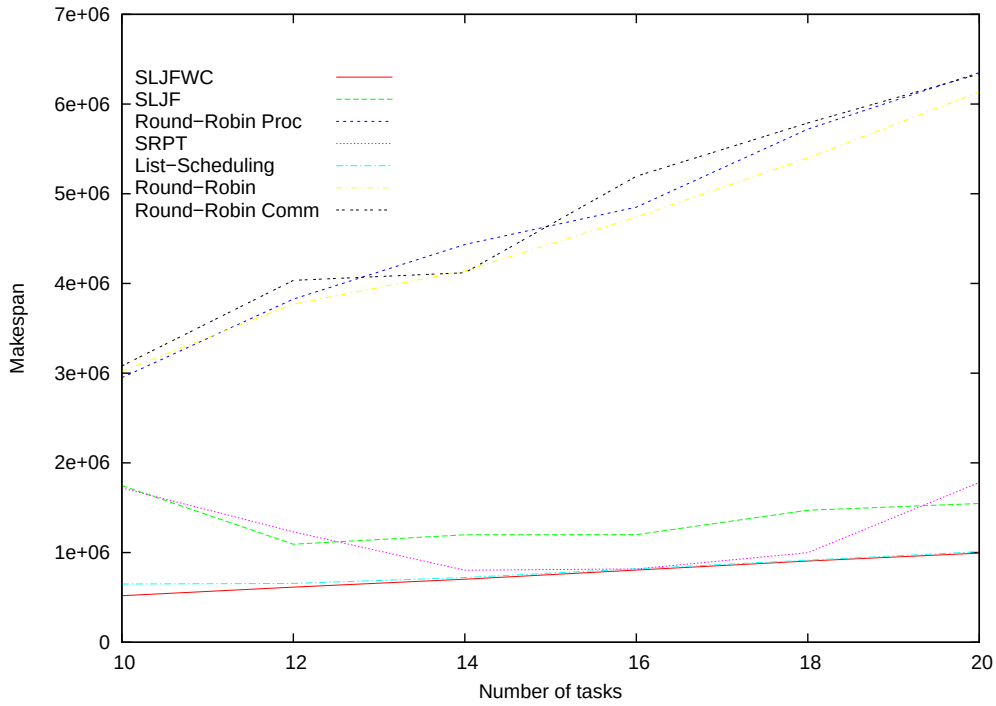


Figure 8: Makespan on a platform with homogeneous slaves.

rectional communication is not possible); also, there is a fixed latency between the initiation of the communication by the sender and the beginning of the reception by the receiver. Still, the model is rather close to the one-port model discussed in this paper.

On-line scheduling– A good survey of on-line scheduling can be found in [25, 22]. Two papers focus on the problem of on-line scheduling for master-slaves platforms. In [17], Leung and Zhao proposed several competitive algorithms minimizing the total completion time on a master-slave platform, with or without pre- and post-processing. In [18], the same authors studied the complexity of minimizing the makespan or the total response time, and proposed some heuristics. However, none of these works take into consideration communication costs.

8 Conclusion

In this paper, we have dealt with the problem of scheduling independent, same-size tasks on master-slave platforms. We enforce the one-port model, and we study the impact of the communications on the design and analysis of the proposed algorithms.

On the theoretical side, we have derived several new results, either for on-line scheduling, or for off-line scheduling with release dates. There are two important directions for future work. First, the bounds on the competitive ratio that we have established for on-line scheduling on communication-homogeneous, and computation-homogeneous platforms, are lower bounds: it would be very interesting to see whether these bounds can be met, and to design the corresponding approximation algorithms. Second, there remains to derive an optimal algorithm for off-line scheduling with release dates on computation-homogeneous platforms.

On the practical side, we have to widen the scope of the MPI experiments. A detailed comparison of all the heuristics that we have implemented needs to be conducted on significantly larger platforms (with several tens of slaves). Such a comparison would, we believe, further demonstrate the superiority of those heuristics which fully take into account the relative capacity of the

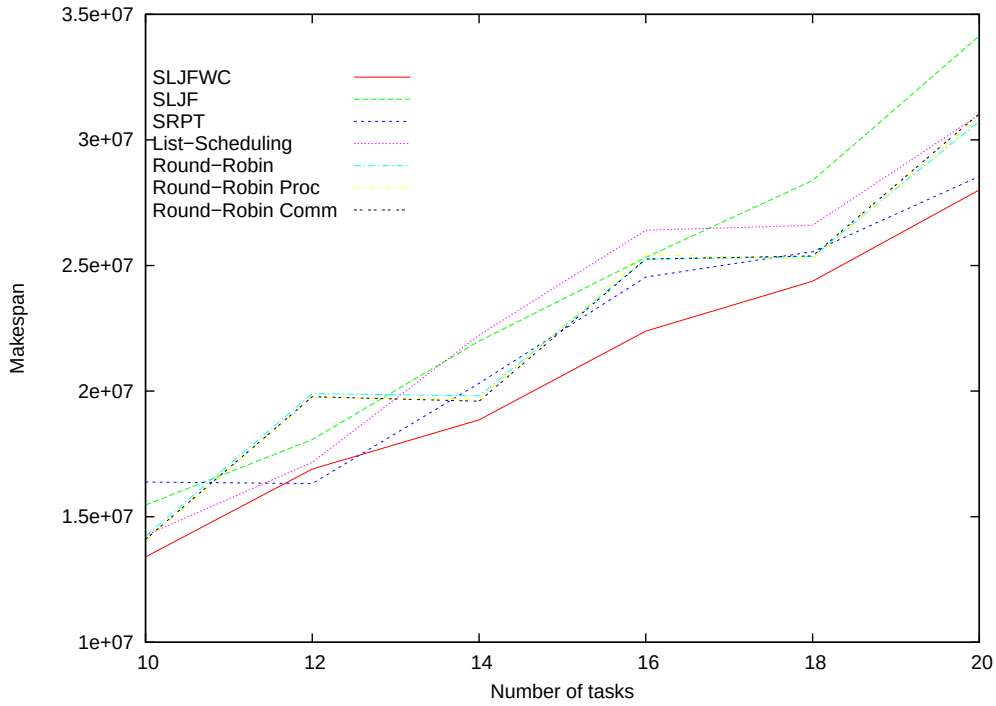


Figure 9: Makespan with release dates.

communication links.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Olivier Beaumont, Arnaud Legrand, and Yves Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *ISCIS XVII, Seventeenth International Symposium On Computer and Information Sciences*, pages 115–119. CRC Press, 2002.

[6] Olivier Beaumont, Arnaud Legrand, and Yves Robert. The master-slave paradigm with heterogeneous processors. *IEEE Trans. Parallel Distributed Systems*, 14(9):897–908, 2003.

[7] 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.

[8] 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.

[9] J. Blazewicz, J.K. Lenstra, and A.H. Kan. Scheduling subject to resource constraints. *Discrete Applied Mathematics*, 5:11–23, 1983.

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

[11] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

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

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

[15] 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.

[16] J.K. Lenstra, R. Graham, E. Lawler, and A.H. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[17] Joseph Y-T. Leung and Hairong Zhao. Minimizing total completion time in master-slave systems, 2004. Available at `http://web.njit.edu/~hz2/papers/masterslave-ieee.pdf`.

[18] Joseph Y-T. Leung and Hairong Zhao. Minimizing mean flowtime and makespan on master-slave systems. *J. Parallel and Distributed Computing*, 65(7):843–856, 2005.

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

[20] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):103–117, 1993.

[21] J. M. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 349–354. IEEE Computer Society Press, 2001.

[22] Irk Pruhs, Jiri Sgall, and Eric Torng. On-line scheduling. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 15.1–15.43. CRC Press, 2004.

[23] C. Roig, A. Ripoll, M. A. Senar, F. Guirado, and E. Luque. Improving static scheduling using inter-task concurrency measures. In T. M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 375–381. IEEE Computer Society Press, 2001.

[24] 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.

[25] J. Sgall. On line scheduling-a survey. In *On-Line Algorithms*, Lecture Notes in Computer Science 1442, pages 196–231. Springer-Verlag, Berlin, 1998.

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

[27] Barbara Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, 12(2):294–299, 1983.

[28] Oliver Sinnen and Leonel Sousa. Communication contention in task scheduling. *IEEE Trans. Parallel Distributed Systems*, 16(6):503–515, 2004.

[29] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):857–871, 1997.