Scheduling tasks sharing files on heterogeneous master-slave platforms

Arnaud Giersch [a,*], Yves Robert [b], Frédéric Vivien [b]

[a] *ICPS/LSIIT, UMR CNRS–ULP 7005*
Parc d'Innovation, Bd Sébastien Brant, BP 10413, 67412 Illkirch Cedex, France

[b] *LIP, UMR CNRS–ENS Lyon–INRIA–UCBL 5668*
École normale supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Abstract

This paper is devoted to scheduling a large collection of independent tasks onto heterogeneous clusters. The tasks depend upon (input) files which initially reside on a master processor. A given file may well be shared by several tasks. The role of the master is to distribute the files to the processors, so that they can execute the tasks. The objective for the master is to select which file to send to which slave, and in which order, so as to minimize the total execution time. The contribution of this paper is twofold. On the theoretical side, we establish complexity results that assess the difficulty of the problem. On the practical side, we design several new heuristics, which are shown to perform as efficiently as the best heuristics in [1,2] although their cost is an order of magnitude lower.

Key words: Scheduling, Heterogeneous clusters, Independent tasks, File-sharing, Heuristics.

1 Introduction

In this paper, we are interested in scheduling independent tasks onto heterogeneous clusters. These independent tasks depend upon files (corresponding to input data, for example), and difficulty arises from the fact that some files may well be shared by several tasks.

* Corresponding author. Tel.: +33 3 90 24 45 42; fax: +33 3 90 24 45 47.
Email addresses: `Arnaud.Giersch@icps.u-strasbg.fr` (Arnaud Giersch), `Yves.Robert@ens-lyon.fr` (Yves Robert), `Frederic.Vivien@ens-lyon.fr` (Frédéric Vivien).

This paper is motivated by the work of Casanova, Legrand, Zagorodnov, and Berman [1,2], who target the scheduling of tasks in APST, the AppLeS Parameter Sweep Template [3]. APST is a grid-based environment whose aim is to facilitate the mapping of applications to heterogeneous platforms. Typically, an APST application consists of a *large* number of independent tasks, with possible input data sharing (see [1,2] for a detailed description of a real-world application). By *large* we mean that the number of tasks is usually at least one order of magnitude larger than the number of available computing resources. When deploying an APST application, the intuitive idea is to map those tasks that depend upon the same files onto the same computational resource, so as to minimize communication requirements. Casanova et al. [1,2] have considered three heuristics designed for completely independent tasks (no input file sharing) that were proposed in [4]. They have modified these three heuristics (originally called *Min-min*, *Max-min*, and *Sufferage* in [4]) to adapt them to the additional constraint that input files are shared between tasks. As was already pointed out, the number of tasks to schedule is expected to be very large, and special attention should be devoted to keeping the cost of the scheduling heuristics reasonably low.

In this paper, we deal with the same scheduling problem as Casanova et al. [1,2]: we assume the existence of a master processor, which serves as the repository for all files. The role of the master is to distribute the files to the processors, so that they can execute the tasks. The objective for the master is to select which file to send to which slave, and in which order, so as to minimize the total execution time. This master-slave paradigm has a fundamental limitation: communications from the master may well become the true bottleneck of the overall scheduling scheme. Allowing for inter-slave communications, and/or for distributed file repositories, should certainly be the subject of future work. However, we believe that concentrating on the simpler master-slave paradigm is a first but mandatory step towards a full understanding of this challenging scheduling problem.

The contribution of this paper is twofold. On the theoretical side, we establish two complexity results that assess the difficulty of the problem:

- The first result shows the NP-completeness of the scheduling problem with a single slave,
- The second result shows the NP-completeness of the scheduling problem with two slaves, in the special case where all tasks and files have same size.

On the practical side, we design several new heuristics, which are shown to perform as efficiently as the best heuristics in [1,2] although their cost is an order of magnitude lower.

The rest of the paper is organized as follows. The next section (Section 2) is de-

voted to the precise and formal specification of our scheduling problem, which we denote as TasksSharingFiles. Next, in Section 3, we state complexity results, which include the two NP-completeness results already mentioned. Then, Section 4 deals with the design of low-cost polynomial-time heuristics to solve the TasksSharingFiles problem. We report some experimental data in Section 5. Finally, we state some concluding remarks in Section 6.

2 Framework

In this section, we formally state the optimization problem to be solved.

2.1 Tasks and files

The problem is to schedule a set of n tasks $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Theses tasks have different sizes: the weight of task T_j is t_j, $1 \leq j \leq n$. There are no dependence constraints between the tasks, so they can be viewed as independent.

However, the execution of each task depends upon one or several files, and a given file may be shared by several tasks. Altogether, there are m files in the set $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$. The size of file F_i is f_i, $1 \leq i \leq m$. We use a bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to represent the relations between files and tasks. The set of nodes in the graph $\mathcal{G}$ is $\mathcal{V} = \mathcal{F} \cup \mathcal{T}$, and each node is weighted by f_i or t_j, depending upon its membership in $\mathcal{F}$ or $\mathcal{T}$. There is an edge $e_{i,j} : F_i \rightarrow T_j$ in $\mathcal{E}$ if and only if task T_j depends on file F_i. Intuitively, files F_i such that $e_{i,j} \in \mathcal{E}$ correspond to some data that is needed for the execution of T_j to begin. The processor that will have to execute task T_j will need to receive all the files F_i such that $e_{i,j} \in \mathcal{E}$ before it can start the execution of T_j. See Figure 1 for a small example, with $m = 9$ files and $n = 13$ tasks. For instance, task T_1 depends upon files F_1 and F_2, and task T_3 depends upon files F_2, F_3, F_4, and F_7.

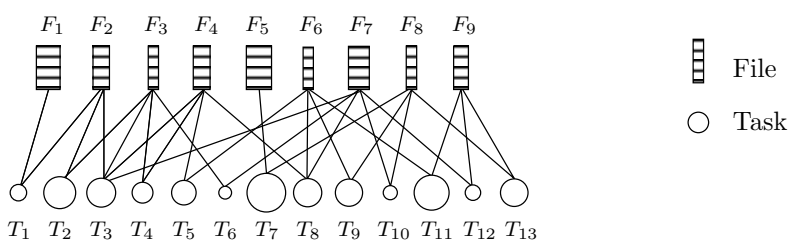

Figure 1. Bipartite graph gathering relations between files and tasks.

To summarize, the bipartite *application graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node in

$\mathcal{V} = \mathcal{F} \cup \mathcal{T}$ is weighted by f_i or t_j, and where edges in $\mathcal{E}$ represent the relations between the files and the tasks, gathers all the information on the application.

2.2 Platform graph

The tasks are scheduled and executed on a master-slave heterogeneous platform. We let $\mathcal{P}$ denote the *platform graph*, which is a fork-graph (see Figure 2) with a master-processor P_0 and p slaves P_i, $1 \leq i \leq p$. Each slave P_q has a (relative) cycle time w_q: it takes $t_j \cdot w_q$ time-units to execute task T_j on processor P_q. We point out that all the results and heuristics of this paper can straightforwardly be extended to the more general case of *inconsistent* execution times, with the terminology of [5]: in that situation, each slave P_q has a different execution time $w_{j,q}$ for each task T_j, and these times are not related; then, we would simply replace all terms $t_j \cdot w_q$ by $w_{j,q}$.

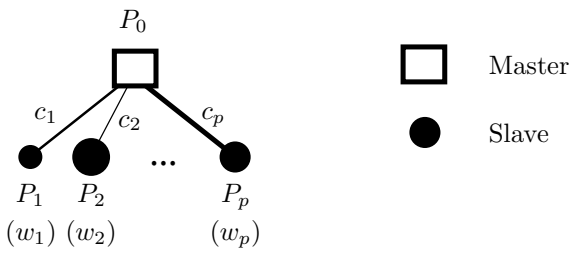

Figure 2. Heterogeneous fork-graph.

The master processor P_0 initially holds all the m files in $\mathcal{F}$. The slaves are responsible for executing the n tasks in $\mathcal{T}$. Before it can execute a task T_j, a slave must have received from the master all the files that T_j depends upon. For communications, we use the one-port model: the master can only communicate with a single slave at a given time-step. We let c_q denote the inverse of the bandwidth of the link between P_0 and P_q, so that $f_i \cdot c_q$ time-units are required to send file F_i from the master to the slave P_q. We assume that communications can overlap computations on the slaves: a slave can process one task while receiving the files necessary for the execution of another task.

Coming back to the example of Figure 1, assume that we have a two-slave schedule such that tasks T_1 to T_6 are executed by slave P_1, and tasks T_7 to T_{13} are executed by slave P_2. Overall, P_1 will receive six files (F_1 to F_4, F_6, and F_7), and P_2 will receive six files (F_4 to F_9). In this schedule, three files (F_4, F_6, and F_7) must be sent to both slaves.

To summarize, we assume a fully heterogeneous master-slave paradigm: slaves have different speeds and links have different capacities. Communications from the master are serial, and may well become the major bottleneck.

2.3 Objective function

The objective is to minimize the total execution time. The execution is terminated when the last task has been completed. The schedule must decide which tasks will be executed by each slave. It must also decide the ordering in which the master sends the files to the slaves. We stress two important points:

- Some files may well be sent several times, so that several slaves can independently process tasks that depend upon these files.
- A file sent to some processor remains available for the rest of the schedule. If two tasks depending on the same file are scheduled on the same processor, the file must only be sent once to that processor.

To decrease the total execution time, we may will try to limit the amount of replicated files. By mapping on a same processor tasks depending on a same file, the communication time will be reduced. But then there is the risk that all tasks are mapped on a single processor. On the contrary, if we try to balance the load between the processors, a lot of communications may be induced. There is a trade-off to be found between these two extreme solutions.

We let TasksSharingFiles$(\mathcal{G}, \mathcal{P})$ denote the optimization problem to be solved.

3 Complexity

Most scheduling problems are known to be difficult [6,7]. However, some particular instances of the TasksSharingFiles optimization problem have a polynomial complexity, while the decision problems associated to other instances are NP-complete. We outline several results in this section, which are all gathered in Figure 3. In Figure 3, the pictographs read as follows: for each of the six case studies, the leftmost diagram represents the application graph, and the rightmost diagram represents the platform graph. We draw objects of different sizes to symbolically represent their heterogeneity. The application graph is made up of files and tasks which all have the same sizes in situations (a), (b), and (c), while this is not the case in situations (d), (e), and (f). Tasks depend upon a single (private) file in situations (a), (b), (d), and (e), which is not the case in situations (c) and (f). As for the platform graph, there is a single slave in situations (d) and (f), and several slaves otherwise. The platform is homogeneous in cases (a) and (e), and heterogeneous in cases (b) and (c). The six situations are discussed in the text below.

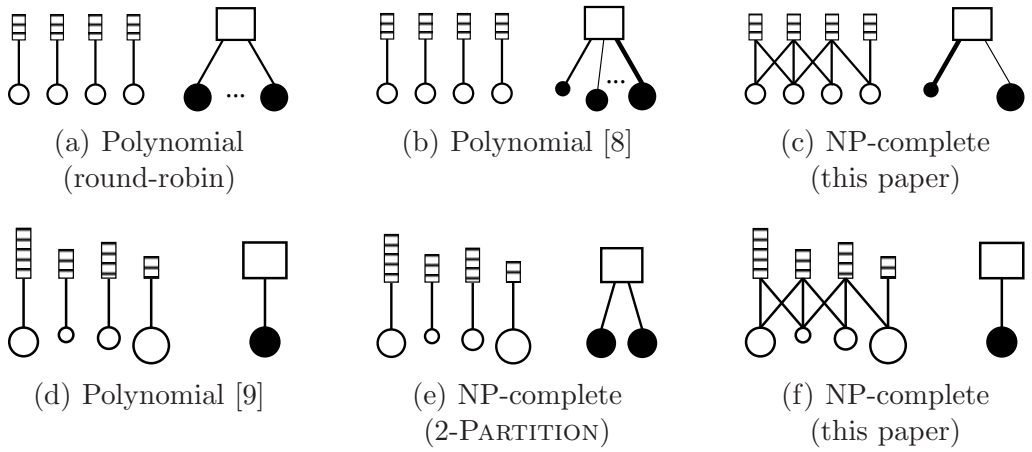

Figure 3. Complexity results for the problem of scheduling tasks sharing files.

3.1 With a single slave

The instance of TasksSharingFiles with a single slave turns out to be more difficult than we would think intuitively. In the very special case where each task depends upon a single non-shared file, i.e., $n = m$ and $\mathcal{E}$ reduces to n edges $e_{i,i} : F_i \to T_i$, the problem can be solved in polynomial time (this is situation (d) of Figure 3). Indeed, it is equivalent to the two-machine flow-shop problem, and the algorithm of Johnson [9] can be used to compute the optimal execution time. According to Johnson's algorithm we first schedule the tasks whose communication time (the time needed to send the file) is smaller than (or equal to) the execution time in increasing order of the communication time. Then we schedule the remaining tasks in decreasing order of their execution time.

The general instance with a single slave, where files are shared between tasks, corresponds to situation (f) of Figure 3. One major result of this paper is to prove the NP-hardness of this instance. Interestingly, this shows that (unless P=NP) there is no polynomial algorithm to extend Johnson's algorithm for general graphs.

The decision problem associated to the general instance of TasksSharing-Files with a single slave can formally be stated as follows:

Definition 1 (TSF1-Dec($\mathcal{G}, \mathcal{P}, p = 1, K$)). Given a bipartite application graph $\mathcal{G}$, a platform $\mathcal{P}$ with a single slave ($p = 1$) and a time bound K, is it possible to schedule all tasks within K time-steps?

Theorem 1. TSF1-Dec($\mathcal{G}, \mathcal{P}, p = 1, K$) is NP-complete.

Proof. Obviously, TSF1-Dec($\mathcal{G}, \mathcal{P}, p = 1, K$) belongs to NP. To prove its

completeness, we use a reduction from MEWC, the Maximum Edge-Weighted Clique problem, which is NP-complete [10]. Consider an arbitrary instance $\mathcal{I}_1$ of MEWC: given a complete edge-weighted graph $G_c = (V_c, E_c, w)$, where $w : E_c \to \mathbb{N}$ is the weight function, a size bound B, where $3 \leq B \leq |V_c|$, and a weight bound $W > 0$, is there a subset S of B vertices such that $\sum_{e \in E_S} w(e) \geq W$? Here, E_S denotes the set of the $B \cdot (B-1)/2$ edges connecting the vertices of S. In other words, can we find B vertices inducing a sub-graph of weight at least W? We point out that the original formulation of MEWC in [10] asks for a subset of *at most* B vertices rather than of *exactly* B vertices, as we do here. However, it is straightforward to see that our formulation remains NP-complete (any polynomial algorithm solving our formulation could be invoked at most $|V|$ times to solve the original formulation).

We construct the following instance $\mathcal{I}_2$ of TSF1-Dec$(\mathcal{G}, \mathcal{P}, p = 1, K)$. We let $\mathcal{F} = V_c \cup \{X\}$ and $\mathcal{T} = E_c \cup \{T_x\}$ (see Figure 4), which defines $\mathcal{V} = \mathcal{F} \cup \mathcal{T}$. There are $m = |V_c| + 1$ files, and $n = |E_c| + 1 = (m-1) \cdot (m-2)/2 + 1$ tasks (the original graph G_c is complete, hence $|E_c| = |V_c| \cdot (|V_c| - 1)/2$).

The size of file X is 1, and the size of each file corresponding to a node in V_c is $f = W \cdot (2B - 1)$. The weight of task T_x is $x = W \cdot (B^2 + 2B - 2)$. Note that $x \geq 0$ because $B \geq 3$. The weight of the task corresponding to an edge $e \in E_c$ is $2W + w(e)$.

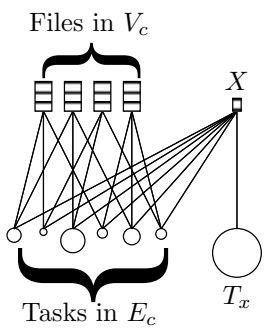

Figure 4. The bipartite application graph used in the proof of Theorem 1, with $|V_c| = 4$.

The relations between tasks and files are defined as follows. First, there is an edge from file X to each task in $\mathcal{T}$. Second, there is an edge from a node (file) $v \in V_c \subset \mathcal{F}$ to a node (task) $e \in E_c \subset \mathcal{T}$ if and only if v was one of the two end-points of edge e in G_c. As a consequence, each edge-task (in $\mathcal{T} \setminus \{T_x\}$) exactly depends upon three files (X, and both end-points of the edge). The computing platform is quite simple: a single slave, with unit communication and computation time: $c_1 = w_1 = 1$. Finally, we define the scheduling bound:

$$K = 1 + x + \sum_{e \in E_c} w(e) + 2W \cdot |E_c|$$

Clearly, the instance $\mathcal{I}_2$ can be constructed in time polynomial in the size of

$\mathcal{I}_1$. Now we have to show that $\mathcal{I}_2$ admits a solution if and only if $\mathcal{I}_1$ has one.

Assume first that $\mathcal{I}_1$ has a solution, i.e., that G_c possesses B vertices inducing a sub-graph whose edge-weight is at least W. Let $\mathcal{C} = \{v_1, v_2, \ldots, v_B\}$ denote these B vertices. The intuitive idea to construct the schedule is as follows: after sending file X, the master sends the B files corresponding to the B nodes in $\mathcal{C}$. Because these files induce a large amount of work, the slave processor will have enough work to process while receiving the other files. The idea is to keep the slave processor active all the time as soon as it has received file X. The bound K is defined accordingly: the first time-unit is spent receiving X, and the rest amounts to the sum of all task weights.

The schedule is defined as follows:

(1) At time-step $t = 0$, file X is sent to the slave.
(2) The master sends the files (corresponding to the nodes) of V_c as soon as possible, i.e., at time $t = 1 + (j-1) \cdot f$ for the j-th file, $1 \leq j \leq |V_c|$ (recall that f is the size of each file in V_c). The first B files sent are chosen to be those in $\mathcal{C}$, in any order. The remaining $|V_c| - B$ files are then sent in any order.
(3) The slave has an execution queue, which it processes greedily, and in FIFO order. At time-step $t = 1$, T_x is available in the queue, and the slave starts its execution. Upon reception of the first file of V_c, no new task is ready. But upon reception of the j-th file of V_c, with $j \geq 2$, there are $j - 1$ new tasks ready for execution: they correspond to all the edges in G_c whose first end-point is the j-th file, and whose other end-point is one of the $j - 1$ files of V_c previously received. These $j - 1$ tasks are inserted at the end of the execution queue, in any order.

We have derived a schedule for instance $\mathcal{I}_2$, but does it match the execution bound K? As already mentioned, this is only possible if the slave is never idle after receiving file X. Let $\mathsf{RC}(j)$ denotes the *receive capacity* of the slave upon reception of the j-th file from V_c: $\mathsf{RC}(j)$ denotes the amount of work that remains to be executed for the current task and those ready in the queue. It corresponds to the time the processor can spend, waiting for a new file, without becoming inactive. Similarly, $\mathsf{RC}(0)$ denotes the receive capacity of the slave upon reception of file X i.e., the time to execute task T_x. Obviously, we would like $\mathsf{RC}(j) \geq f$ for all $j \geq 0$: this would allow the slave to receive a new file without becoming idle.

Initially, owing to task T_x, we have $\mathsf{RC}(0) = x$. Let E_j denote the set of the tasks from E_c that only depend on the first j files from V_c sent to the slave.

We have $\mathsf{RC}(1) = x - f$ (the first file does not grant any work), and

$$\mathsf{RC}(j) = x + \sum_{e \in E_j} w(e) + 2W \cdot |E_j| - f \cdot j$$

for all $j \geq 2$. Indeed, this quantity is the sum of the execution times of the tasks in $E_j \cup \{T_x\}$, minus the time spent to send the first j files. We want to show that $\mathsf{RC}(j) \geq f$ for all j, $0 \leq j \leq |V_c|$. We have $\mathsf{RC}(0) - f = x - f \geq x - 2f = \mathsf{RC}(1) - f$. Furthermore, $x - 2f = W \cdot (B^2 - 2B) \geq 0$, since $B \geq 3$. So $\mathsf{RC}(0) \geq f$ and $\mathsf{RC}(1) \geq f$. For $j \geq 2$, $|E_j| = j \cdot (j-1)/2$, and $\mathsf{RC}(j) - f = \sum_{e \in E_j} w(e) - W + h(j)$, where $h(j) = W \cdot (j - B)^2$. The minimum of $h(j)$ is zero, and is obtained for $j = B$. But due to the choice of the first B files sent to the slave, $\sum_{e \in E_B} w(e) \geq W$, hence $\mathsf{RC}(B) - f \geq 0$. For $j \neq B$, $h(j) \geq h(B-1) = h(B+1) = W$ and $\mathsf{RC}(j) - f \geq h(j) - W \geq 0$. Altogether, this concludes the proof that the total execution of the schedule is equal to K, hence a solution to $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution. We have a schedule with executes within $K = 1 + x + \sum_{e \in E_c} w(e) + 2W \cdot |E_c|$ time-units. But K is equal to one plus the sum of the task weights. Because the slave processor is idle until file X has been sent, necessarily the first file sent is X, and this emission lasts one time-unit. After the first time-step, the slave processor must be kept busy all the time. Letting E_j be the set of the tasks from E_c that only depend of the first j files of V_c sent to the slave, we must have as previously, $\mathsf{RC}(j) \geq f$ for all $1 \leq j \leq |V_c|$. For $j \geq 2$, we know that $\mathsf{RC}(j) \leq x + \sum_{e \in E_j} w(e) + 2W \cdot |E_j| - f \cdot j$. We had an equality before, but maybe the schedule did not send the files as soon as possible, hence the inequality here. Taking $j = B$ we derive just as before that $\mathsf{RC}(B) - f \leq \sum_{e \in E_B} w(e) - W$. Because the slave is never idle after receiving the B-th file, we have $\mathsf{RC}(B) - f \geq 0$, and we derive that $\sum_{e \in E_B} w(e) \geq W$. The first B files sent to the slave provide a solution to $\mathcal{I}_1$. □

3.2 With two slaves

With several slaves, some problem instances have polynomial complexity. First of all, a greedy round-robin algorithm is optimal in situation (a) of Figure 3: each task depends upon a single non-shared file, all tasks and files have the same size, and the fork platform is homogeneous. If we keep the same hypotheses for the application graph but move to heterogeneous slaves (situation (b) of Figure 3), the problem remains polynomial, but the optimal algorithm becomes complicated: see [8] for a description and proof.

The decision problem associated to the general instance of TasksSharingFiles with two slaves, writes as follows:

Definition 2 (TSF2-Dec($\mathcal{G}, \mathcal{P}, p = 2, K$)). Given a bipartite application graph $\mathcal{G}$, a heterogeneous platform $\mathcal{P}$ with two slaves ($p = 2$), and a time bound K, is it possible to schedule all tasks within K time-steps?

Clearly, TSF2-Dec is NP-complete, even if there are no files at all: in that case, TSF2-Dec reduces to the scheduling of independent tasks on a two-processor machine, which itself reduces to the 2-Partition problem [11] as the tasks have different sizes. This corresponds to situation (e) in Figure 3, where we do not even need the private files. However, this NP-completeness result does not hold in the strong sense: in a word, the size of the tasks plays a key role in the proof, and there are pseudo-polynomial algorithms to solve TSF2-Dec in the simple case when there are no files (see the pseudo-polynomial algorithm for 2-Partition in [11]).

The following theorem states an interesting result: in the case where all files and tasks have unit size (i.e., $f_i = t_j = 1$), the TSF2-Dec remains NP-complete. Note that in that case, the heterogeneity only comes from the computing platform. This corresponds to situation (c) in Figure 3.

Definition 3 (TSF2-Equal-Dec($\mathcal{G}, \mathcal{P}, p = 2, f_i = t_j = 1, K$)). Given a bipartite application graph $\mathcal{G}$ such that $f_i = t_j = 1$ for all tasks and files, a heterogeneous platform $\mathcal{P}$ with two slaves ($p = 2$), and a time bound K, is it possible to schedule all tasks within K time-steps?

Theorem 2. TSF2-Equal-Dec($\mathcal{G}, \mathcal{P}, p = 2, f_i = t_j = 1, K$) is NP-complete.

Proof. Obviously, TSF2-Equal-Dec($\mathcal{G}, \mathcal{P}, p = 2, f_i = t_j = 1, K$) belongs to NP. To prove its completeness, we use a reduction from Clique, which is NP-complete [11]. Consider an arbitrary instance $\mathcal{I}_1$ of Clique: given a graph $G_c = (V_c, E_c)$, and a bound B, is there a clique in G_c (i.e., a fully connected sub-graph) of size B? Without loss of generality, we assume that $|V_c| \geq 9$ and $6 \leq B \cdot (B - 1) < |E_c|$.

We construct the following instance $\mathcal{I}_2$ of TSF2-Equal-Dec($\mathcal{G}, \mathcal{P}, p = 2$, $f_i = t_j = 1, K$). We let $\mathcal{F} = V_c \cup X$ and $\mathcal{T} = E_c \cup \{T_y\}$ (see Figure 5), which defines $\mathcal{V} = \mathcal{F} \cup \mathcal{T}$. Here, X is a collection of $x = |X|$ additional files, so there is a total of $|V_c| + x$ files, one per node in the original graph G_c and one per new file in X. As for tasks, there are as many tasks as edges in the original graph G_c, plus an additional task T_y.

The relations between tasks and files are defined as follows. First, there is an edge from each file in $\mathcal{F}$ to task T_y; as a consequence, the slave processor that will execute T_y will need to have received all the files in $\mathcal{F}$ from the master before it can begin the execution of T_y. Second, there is an edge from a node (file) $v \in V_c \subset \mathcal{F}$ to a node (task) $e \in E_c \subset \mathcal{T}$ if and only if v was one of the

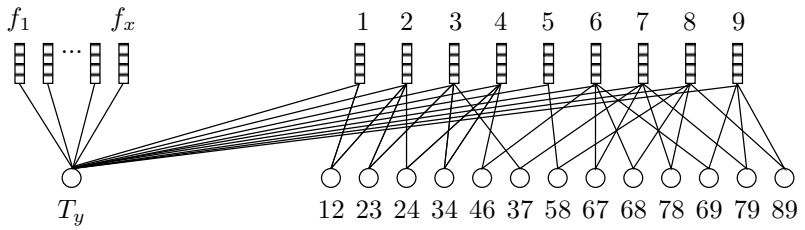

Figure 5. The bipartite application graph used in the proof of Theorem 2.

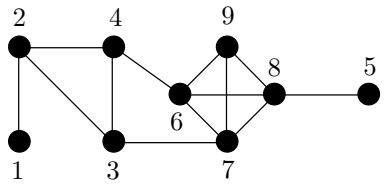

Figure 6. The original graph G_c used to build the bipartite graph of Figure 5.

two end-points of edge e in G_c. In the rightmost part of Figure 5, the bipartite graph has been obtained from the original graph G_c shown in Figure 6. The files are the nodes in G_c, and the tasks are the edges in G_c. This explains why each task (edge) exactly depends upon two files (the end-points of the edge). We see that G_c has a clique of size $B = 4$ (nodes 6 to 9).

As specified in the problem, all files and tasks have unit size. To complete the description of the application, we let $s = B \cdot (B-1)/2$, $r = |E_c| - s$ (note that $s < r$ by hypothesis), and we define $x = (3r-1) \cdot |V_c| - 2B + 2$. We check that $x \geq 1$: indeed, $r \geq 4$ and $|V_c| \geq B$; we derive $x \geq 9B + 2$.

There remains to describe the computing platform. The characteristics of the two slave processors are: $w_1 = 3 \cdot |V_c|$, $w_2 = (3 \cdot (r+1) \cdot |V_c| - 4)/s$, $c_1 = 1$, and $c_2 = 2$. Note that $w_2 > w_1$, because $w_2 - w_1 = 3 \cdot (r/s - 1) \cdot |V_c| + (3 \cdot |V_c| - 4)/s > 0$. Thus, $w_2 > 2$. Finally, we define the scheduling bound:

$$K = 2 + 3 \cdot (r+1) \cdot |V_c| = 6 + s \cdot w_2.$$

Clearly, the instance $\mathcal{I}_2$ can be constructed in time polynomial in the size of $\mathcal{I}_1$. Now we have to show that $\mathcal{I}_2$ admits a solution if and only if $\mathcal{I}_1$ has one.

Assume first that $\mathcal{I}_1$ has a solution, i.e., that G_c possesses a clique of size B. Let $\mathcal{C} = \{v_1, v_2, \ldots, v_B\}$ denote the B vertices in the clique of G_c. The intuitive idea is the following: after sending to slave P_2 the B files corresponding to the B nodes in $\mathcal{C}$, P_2 will be able to process the s tasks that correspond to the edges connecting the nodes of $\mathcal{C}$ without receiving any extra file from the master. The schedule is defined as follows:

- First, at time-steps $t = 0$ and $t = 1$, two files are sent by the master to P_1.

These two files are chosen so that they correspond to any two nodes v_a and v_b of V_c that are connected in G_c (i.e., the edge (v_a, v_b) belongs to E_c) and that do not both belong to the clique $\mathcal{C}$. Note that such an edge exists, as the number of edges with their two end-points in $\mathcal{C}$ if $B \cdot (B-1)/2 < |E_c|$ (by hypothesis). At time-step $t = 2$, P_1 is able to start the execution of the task that corresponds to the edge (v_a, v_b). P_1 terminates this task at time-step $2 + w_1 = 2 + 3 \cdot |V_c|$.

- Next, the B files that correspond to the clique $\mathcal{C}$ are sent to P_2. As soon as it has received two files, P_2 can start executing one task (the two files correspond to two connected nodes, therefore the task that represents the edge between them is ready for execution). P_2 has received the B files at time-step $2c_1 + B \cdot c_2 = 2 + 2B$, i.e., before it completes the execution of the first task, at time-step $2c_1 + 2c_2 + w_2 = 6 + w_2 > 6 + w_1 = 6 + 3 \cdot |V_c| \geq 6 + 3B$, because $B \leq |V_c|$. Therefore, P_2 can process the s tasks corresponding to edges in the clique $\mathcal{C}$ without interruption (i.e., without waiting to receive more files), until time-step $2c_1 + 2c_2 + s \cdot w_2 = 6 + s \cdot w_2 = K$.
- Finally, after sending the B files to P_2, all files but two are sent to P_1: we do not re-send the first two files, but we send all the others, i.e., $|V_c| - 2 + x$ files. We send the $|V_c| - 2$ files corresponding to nodes in V_c before the x files corresponding to nodes in X. When P_1 terminates its first task, at time-step $2 + 3 \cdot |V_c|$, it has already received the first $|V_c| - 2$ files (the last one is received at time-step $2c_1 + B \cdot c_2 + (|V_c| - 2) \cdot c_1 = |V_c| + 2B$). P_1 can process the r tasks corresponding to edges in G_c that do not belong to the clique $\mathcal{C}$ without interruption, until time-step $2c_1 + r \cdot w_1 = K - w_1$. At that time-step, P_1 has just received the x last files, because $(|V_c| + x) \cdot c_1 + B \cdot c_2 = K - w_1$. P_1 processes then the last task T_y, and the scheduling terminates within K times-steps.

We have derived a valid solution to our scheduling instance $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution. We proceed in several steps:

(1) Necessarily, P_1 executes task T_y. Otherwise, P_2 would execute it, but T_y requires $|V_c| + x$ files, and the time needed by P_2 would be at least $(|V_c| + x) \cdot c_2 + w_2 = 2 \cdot (K - w_1 - 2B) + w_2 > 2 \cdot (K - 5 \cdot |V_c|) > K$ (because $K \geq 15 \cdot |V_c|$), a contradiction.

(2) P_1 cannot execute more than $(K - 2c_1)/w_1 = r + 1$ tasks, because it must have received two files before it can start to process its first task.

(3) All files sent by the master after time-step $K - w_1$ are useless, because the tasks that they might free for execution will not be terminated at time-step K, neither by P_1 nor by P_2 (remember that $w_2 > w_1$). Because P_1 executes T_y, it receives $|V_c| + x$ files. But $K - w_1 = (|V_c| + x) \cdot c_1 + B \cdot c_2$, so that P_2 cannot receive more than B tasks from the master.

(4) P_2 cannot execute more than s tasks, because $(K - 2c_2)/w_2 = (K - 6)/w_2 + 2/w_2 = s + 2/w_2 < s + 1$.

Overall, a total of $r + s + 1$ tasks are executed. Since P_1 cannot execute more than $r + 1$ tasks, and P_2 more than s tasks, they do execute $r + 1$ and s tasks respectively. But P_2 executes s tasks and receives no more than B files: these files define a clique of size B in G_c, thereby providing a solution to $\mathcal{I}_1$. □

Finally, we have shown that the decision problem associated with TasksSharingFiles is NP-complete, even in the simple cases where:

(1) there is a single slave, but tasks and files have heterogeneous sizes;
(2) tasks and files have unitary size, but the platform is composed of two heterogeneous processors connected with links of different bandwidths.

At the time of this writing, we do not know the complexity of the problem instance where the platform is homogeneous and tasks and files have unitary size. We do not even know the complexity when there is a single slave (and homogeneous tasks and sizes).

In the general version of the problem, everything is heterogeneous: the sizes of the tasks and of the files are different, the slave processors have different speeds and are connected to the master with links of different bandwidths. Therefore, in the following, we design polynomial heuristics to solve the TasksSharingFiles problem, and we assess their performance trough extensive simulations.

4 Heuristics

In this section, we first recall the three heuristics used by Casanova et al. [1,2]. Next we introduce several new heuristics, whose main characteristic is a lower computational complexity.

4.1 Reference heuristics

Because our work was originally motivated by the paper of Casanova et al. [1,2], we have to compare our new heuristics to those presented by these authors, which we call *reference heuristics*. We start with a description of these reference heuristics.

4.1.1 Structure of the heuristics.

All the reference heuristics are built on the model presented by Algorithm 1: while there remain tasks to be scheduled, an objective function is evaluated

Algorithm 1. Structure of reference heuristics.

```
1  S ← T                    ▷ S is the set of the tasks that remain to be scheduled
2  while S ≠ ∅ do
3      foreach processor P_i do
4          foreach task T_j ∈ S do
5              Evaluate OBJECTIVE(T_j, P_i)
6          end
7      end
8      Pick the "best" couple of a task T_j ∈ S and a processor P_i according to
       OBJECTIVE(T_j, P_i)
9      Schedule T_j on P_i as soon as possible
10     Remove T_j from S
11 end
```

for all pairs of a task (which remains to be scheduled) and a processor. The task that will actually be scheduled, as well as the target processor, are chosen according to the values of this objective function.

4.1.2 Objective function.

For all the heuristics, the objective function is the same. OBJECTIVE(T_j, P_i) is indeed the minimum completion time (MCT) of task T_j if mapped on processor P_i. Of course, the computation of this completion time takes into account:

(1) the files required by T_j that are already available on P_i (we assume that any file that once was sent to processor P_i is still available and do not need to be resent);
(2) the time needed by the master to send the other files to P_i, knowing what communications are already scheduled;
(3) the tasks already scheduled on P_i.

4.1.3 Chosen task.

The heuristics only differ by the definition of the "best" couple (T_j, P_i). More precisely, they only differ by the definition of the "best" task. Indeed, the "best" task T_j is always mapped on its most favorable processor (denoted $P(T_j)$), i.e., on the processor which minimizes the objective function:

$$\text{OBJECTIVE}(T_j, P(T_j)) = \min_{1 \leq q \leq p} \text{OBJECTIVE}(T_j, P_q)$$

Here is the criterion used for each reference heuristic:

Min-min: the "best" task T_j is the one minimizing the objective function when mapped on its most favorable processor; shortest tasks are scheduled

first to avoid gaps at the beginning of the schedule:

$$\textsc{Objective}(T_j, P(T_j)) = \min_{T_k \in \mathcal{S}} \min_{1 \le l \le p} \textsc{Objective}(T_k, P_l)$$

Max-min: the "best" task is the one whose objective function, on its most favorable processor, is the largest; the idea is that a long task scheduled at the end would delay the end of the whole execution:

$$\textsc{Objective}(T_j, P(T_j)) = \max_{T_k \in \mathcal{S}} \min_{1 \le l \le p} \textsc{Objective}(T_k, P_l)$$

Sufferage: the "best" task is the one which will be the most penalized if not mapped on its most favorable processor but on its second most favorable processor, i.e., the "best" task is the one maximizing:

$$\min_{P_q \neq P(T_j)} \textsc{Objective}(T_j, P_q) - \textsc{Objective}(T_j, P(T_j))$$

with

$$\textsc{Objective}(T_j, P(T_j)) = \min_{1 \le l \le p} \textsc{Objective}(T_j, P_l)$$

Sufferage II and **Sufferage X:** these are refined version of the *Sufferage* heuristic. The penalty of a task is no more computed using the second most favorable processor but by considering the first processor inducing a significant increase in the completion time. See [1,2] for details.

4.1.4 Computational complexity.

The loop on Step 3 of the reference heuristics computes the objective function for any pair of processor and task. For each processor, this computation has a worst case complexity of $O(|\mathcal{S}| + |\mathcal{E}|)$, where $\mathcal{E}$ is the set of the edges representing the relations between files and tasks (see Section 2.1). Hence, the overall complexity of the heuristics is: $O(p \cdot n \cdot (n + |\mathcal{E}|))$. The complexity is even worse for *Sufferage II* and *Sufferage X*, as the processors must be sorted for each task, leading to a complexity of $O(p \cdot n \cdot (n \cdot \log p + |\mathcal{E}|))$.

4.2 Structure of the new heuristics

When designing new heuristics, we took special care to decreasing the computational complexity. The reference heuristics are very expensive for large problems. We aimed at designing heuristics which are an order of magnitude faster, while trying to preserve the quality of the schedules produced.

In order to avoid the loop on all the pairs of processors and tasks of Step 3 of the reference heuristics, we need to be able to pick (more or less) in constant

Algorithm 2. Structure of the new heuristics.

```
foreach processor P_i do
    foreach task T_j ∈ 𝒯 do
        Evaluate Objective(T_j, P_i)
    end
    Build the list L(P_i) of the tasks sorted according to the value of
    Objective(T_j, P_i)
end
while there remain tasks to schedule do
    foreach processor P_i do
        Let T_j be the first unscheduled task in L(P_i)
        Evaluate CompletionTime(T_j, P_i)
    end
    Pick the couple of a task T_j and a processor P_i minimizing
    CompletionTime(T_j, P_i)
    Schedule T_j on P_i as soon as possible
    Mark T_j as scheduled
end
```

time the next task to be scheduled. Thus we decided to sort the tasks *a priori* according to an objective function. However, since our platform is heterogeneous, the task characteristics may vary from one processor to the other. For example, Johnson's [9] criterion which splits the tasks into two sets (communication time smaller than, or greater than, computation time) depends on the processors characteristics. Therefore, we compute one sorted list of tasks for each processor. Note that this sorted list is computed *a priori* and is not modified during the execution of the heuristic.

Once the sorted lists are computed, we still have to map the tasks to the processors and to schedule them. The tasks are scheduled one-at-a-time. When we want to schedule a new task, on each processor P_i we evaluate the completion time of the first task (according to the sorted list) which has not yet been scheduled. Then we pick the pair task/processor with the lowest completion time. This way, we obtain the structure of heuristics presented by Algorithm 2.

We still have to define the objective functions used to sort the tasks. This is the object of the next section.

4.3 The objective functions

The intuition behind the following six objective functions is quite obvious:

Duration: we just consider the overall execution time of the task as if it was

the only task to be scheduled on the platform:

$$\textsc{Objective}(T_j, P_i) = t_j \cdot w_i + \sum_{e_{k,j} \in \mathcal{E}} f_k \cdot c_i.$$

The tasks are sorted by increasing objectives, which mimics the *Min-min* heuristic.

Payoff: when mapping a task, the time spent by the master to send the required files is payed by all the (waiting) processors as the master can only send files to a single slave at a time, but the whole system gains the completion of the task. Hence, the following objective function encodes the payoff of scheduling the task T_j on the processor P_i:

$$\textsc{Objective}(T_j, P_i) = \frac{t_j}{\sum_{e_{k,j} \in \mathcal{E}} f_k}.$$

The tasks are sorted by decreasing payoffs. Note that the actual objective function to compute the payoff of scheduling task T_j on processor P_i would be: $\textsc{Objective}(T_j, P_i) = t_j \cdot w_i / (\sum_{e_{k,j} \in \mathcal{E}} f_k \cdot c_i)$; as the factors w_i and c_i do not change the relative *order* of the tasks on a given processor, we just dropped these factors. Furthermore, the order of the tasks does not depend on the processor, so only one sorted list is required with this objective function.

Advance: to keep a processor busy, we need to send it all the files required by the next task that it will process, before it ends the execution of the current task. Hence the execution of the current task must be larger than the time required to send the files. We tried to encode this requirement by considering the difference of the computation- and communication-time of a task, i.e., the advance earned due to the execution of this task. Hence the objective function:

$$\textsc{Objective}(T_j, P_i) = t_j \cdot w_i - \sum_{e_{k,j} \in \mathcal{E}} f_k \cdot c_i.$$

The tasks are sorted by decreasing objectives.

Johnson: we sort the tasks on each processor as Johnson does for a two-machine flow shop (see Section 3.1).

Communication: as the communications may be a bottleneck we consider the overall time needed to send the files a task depends upon as if it was the only task to be scheduled on the platform:

$$\textsc{Objective}(T_j, P_i) = \sum_{e_{k,j} \in \mathcal{E}} f_k.$$

The tasks are sorted by increasing objectives, like for *Duration*. As for *Payoff*, the sorted list is processor independent, and only one sorted list is required with this objective function. This simple objective function is

inspired by the work in [8] on the scheduling of homogeneous tasks on an heterogeneous platform.

Computation: symmetrically, we consider the execution time of a task as if it was not depending on any file:

$$\textsc{Objective}(T_j, P_i) = t_j.$$

The tasks are sorted by increasing objectives. Once again, the sorted list is processor independent.

4.4 Additional policies

In the definition of the previous objective functions, we did not take into account the fact that the files are potentially shared between the tasks. Some of them will probably be already available on the processor where the task is to be scheduled, at the time-step we would try to schedule it. Therefore, on top of the previous objective functions, we add the following additional policies. The goal is (to try) to take file sharing into account.

Shared: when a file is sent to a processor, it is beneficial to all tasks depending upon it. We try to express this idea by using, in the objective functions, weighted sizes for the files. The weighted size of a file is obtained by dividing its size by the number of tasks that are dependent upon the file. For example, the objective function for *Duration+shared* is

$$t_j \cdot w_i + \sum_{e_{k,j} \in \mathcal{E}} \frac{f_k}{|\{T_l \mid e_{k,l} \in \mathcal{E}\}|} \cdot c_i.$$

Readiness: for a given processor P_i, and at a given time, the "ready" tasks are the ones whose files are already all on P_i. Under the *Readiness* policy, if there is any ready task on processor P_i at Step 9 of the heuristics, we pick one ready task instead of the first unscheduled task in the sorted list $L(P_i)$.

Locality: in order to try to decrease the amount of file replication, we (try to) avoid mapping a task T_j on a processor P_i if some of the files that T_j depends upon are already present on another processor. To implement this policy, we modify Step 9 of the heuristics. Indeed, we no longer consider the first unscheduled task in $L(P_i)$, but the next unscheduled task which does not depend on files present on another processor. If we have scanned the whole list, and if there remains some unscheduled tasks, we restart from the beginning of the list with the original task selection scheme (first unscheduled task in $L(P_i)$).

Finally, we obtain as many as 44 variants, since any combination of the three additional policies may be used for the six base objective functions, except for

Shared which does not impact *Computation*.

4.5 Computational complexity

Overall, there are $|\mathcal{E}|$ dependence relations between tasks and files. Thus, computing the value of an objective function for all tasks on all processors has a cost of $O(p \cdot (n + |\mathcal{E}|))$, except for heuristic *Computation* for which the cost is $O(p \cdot n)$, as the relations between tasks and files are not considered. So the construction of all the sorted lists has a cost of $O(p \cdot n \cdot \log n + p \cdot |\mathcal{E}|)$ for heuristics *Duration*, *Advance*, and *Johnson* (p lists), of $O(n \cdot \log n + |\mathcal{E}|)$ for heuristics *Payoff* and *Communication* (a single list), and of $O(n \cdot \log n)$ for heuristic *Computation* (a single list). If we denote by ΔT, one plus the maximum number of files that a task depends upon, the execution of the loop at Step 7 of the heuristics (see Algorithm 2) has an overall cost of $O(p \cdot n \cdot \Delta T)$. Note that $n \cdot \Delta T \geq |\mathcal{E}|$. Hence the overall execution time of the heuristics is:

$$O(p \cdot n \cdot (\log n + \Delta T))$$

for heuristics using several lists (*Duration*, *Advance*, and *Johnson*), and

$$O(n \cdot \log n + p \cdot n \cdot \Delta T)$$

for the others (*Payoff*, *Communication*, and *Computation*). We have replaced the term $n + |\mathcal{E}|$ in the complexity of the reference heuristics by the term $\log n + \Delta T$. The experimental results will assert the gain in complexity.

Note that all the additional policies can be implemented without increasing the complexity of the base cases. It is obvious for *Shared*. *Readiness* can be implemented without overhead if we maintain, for all tasks on all processors, a counter of the number of files that are missing for the task on the processor. Each time a file is sent, this counter is updated. When a counter comes to zero, the corresponding task is moved into the set of tasks that are ready for the processor. For *Locality*, one just have to remember, for each task, where the files it depends upon have already been sent: on a single processor (we keep its number), or already spread over several processors. When a file is sent to some slave, this flag is updated for all tasks depending upon the file. In each case, for all tasks on a given processor, there are at most $|\mathcal{E}|$ updates. The additional cost of the *Readiness* and *Shared* policies is thus of $O(p \cdot |\mathcal{E}|)$, which is completely absorbed by the overall complexity of the heuristics.

5 Experimental results

In order to compare our heuristics and the reference heuristics, we have simulated their executions on randomly built platforms and graphs. We have conducted a very large number of experiments, which we summarize in this section.

5.1 Experimental platforms

Processors: we have recorded the cycle time of the different computers used in our laboratories (in Lyon and Strasbourg). From this set of values, we randomly pick values whose difference with the mean value was less than the standard deviation. This way we define a realistic and heterogeneous set of *20 processors*.

Communication links: the *20 communication links* between the master and the slave are built along the same principles as the set of processors.

Communication to computation cost ratio: The absolute values of the communication link bandwidths or of the processors speeds have no meaning (in real life they are application dependent and must be pondered by application characteristics). We are only interested by the relative values of the processors speeds, and of the communication links bandwidths. Therefore, we normalize processor and communication average characteristics. Also, we arbitrarily impose the communication-to-computation cost ratio, so as to model three main types of problems: computation intensive (ratio=0.1), communication intensive (ratio=10), and intermediate (ratio=1).

5.2 Application graphs

We run the heuristics on the following four types of application graphs. In each case, the sizes of the files and tasks are randomly and uniformly taken between 0.5 and 5.

The graphs are schematically represented on Figure 7.

Forks: each graph contains 100 fork graphs, where each fork graph is made up of 20 tasks depending on a single and same file (fig. 7(a)).

Two-one: each task depends on exactly two files: one file which is shared with some other tasks, and one unshared file (fig. 7(b)).

Partitioned: the graph is divided into 20 chunks of 100 tasks, and in each chunk each task randomly depends on 1 up to 10 files. The whole graph contains at least 20 different connected components (fig. 7(c)).

Random: each task randomly depends on 1 up to 50 files (fig. 7(d)).

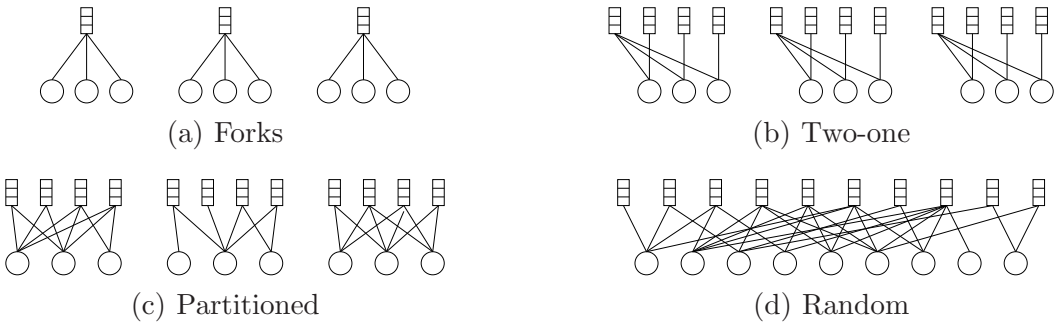

Figure 7. The four type of application graphs used in the simulations.

Our objective is to use graphs *representative* of a large application class. The fork graphs represent embarrassingly parallel applications. The two-one graphs come from the original papers by Casanova et al [2,1]. The partitioned graphs deal with applications encompassing some regularity. The random graphs are for totally irregular applications. Each of our graphs contains 2,000 tasks and 2,500 files, except for the fork graphs which also contain 2,000 tasks but only 100 files.

In order to avoid any interference between the graph characteristics and the communication-to-computation cost ratio, we normalize the sets of tasks and files so that the sum of the file sizes equals the sum of the task sizes times the communication-to-computation cost ratio.

5.3 Results

Table 1 summarizes all the experiments. In this table, we report the performance of the best ten heuristics, together with their cost (i.e., their CPU time). This is a summary of 12,000 random tests (1,000 tests over all four graph types and three communication-to-computation cost ratios). Each test involves 49 heuristics (5 reference heuristics and 44 combinations for our new heuristics). For each test, we compute the ratio of the performance of all heuristics over the best heuristic, which gives us a *relative performance*. The best heuristic differs from test to test, which explains why no heuristic in Table 1 can achieve an average relative performance exactly equal to 1. In other words, the best heuristic is not always the best of each test, but it is closest to the best of each test on average. The optimal relative performance of 1 would be achieved by picking, for any of the 12,000 tests, the best heuristic for this particular case. (For each test, the relative cost is computed along the same guidelines, using the fastest heuristic.) Note that we report extensive simulation results in [12].

We see that *Sufferage* gives the best results: on average, it is within 11% of the relative optimal of 1. The next nine heuristics closely follow: they are within 13% to 14.7% of the relative optimal. Out of these nine heuristics,

only *Min-min* is a reference heuristic. *Max-min* is almost the worst heuristic. This can be explained as follows: in the beginning, this heuristic advantages tasks that have no files on any slave, generating lots of communications, and thereby delaying the execution of the following tasks. Despite the additional time spent to compute the schedule, *Sufferage II* and *Sufferage X* do not achieve as good performance as *Sufferage*. The variants of *Sufferage* base their scheduling choices on a prediction of what would happen if these choices were not selected. This prediction does not take into account the possible file transfers due to the scheduling of other tasks. Thus it is not surprising that *Sufferage II* and *Sufferage X*, which try to make predictions in a longer term, make bigger mistakes, and finally achieve worst results.

Concerning our new heuristics, we can see that the performance of those appearing in Table 1 closely follows the performance of *Min-min*. Furthermore, the standard deviations are lower for our heuristics, which reflects a greater stability of the results.

On the average, *Duration*, *Computation*, and *Payoff* (along with *Readiness*) achieve the best performances. *Communication* lags well behind. However, the results for *Computation* and *Communication* must be nuanced. The performance of *Computation* degrades as the communication-to-computation cost ratio increases, while it is the inverse for *Communication*. This is not surprising, when looking at the definition of these heuristics. *Computation* only uses the computation time of the tasks, hence is more adapted to problems that are computation intensive. A similar explanation holds for *Communication*.

Table 1
Relative performance and cost of the best ten heuristics.

Heuristic	Relative performance	Standard deviation	Relative cost	Standard deviation
Sufferage	1.110	0.1641	376.7	153.4
Min-min	1.130	0.1981	419.2	191.7
Computation+readiness	1.133	0.1097	1.569	0.4249
Duration+locality+readiness	1.133	0.1295	1.499	0.4543
Duration+readiness	1.133	0.1299	1.446	0.3672
Payoff+shared+readiness	1.138	0.1260	1.496	0.6052
Payoff+readiness	1.139	0.1266	1.246	0.2494
Payoff+shared+locality+readiness	1.145	0.1265	1.567	0.5765
Payoff+locality+readiness	1.145	0.1270	1.318	0.2329
Computation+locality+readiness	1.147	0.1234	1.618	0.4749

Duration, which combines both approaches, is more stable across the different communication-to computation cost ratios.

Advance, *Payoff*, and *Johnson* are all based on the same observation: one should try to maximize the overlap of the communications by the computations. They however achieve performances that are very different. In any case, *Advance* and its variants perform badly in comparison with the other heuristics. Among those three heuristics, *Payoff* is the one with the best performance. It seems that *Advance* tends to favor tasks with large communication times. This observation is verified by the fact that it is improved by the *Shared* policy, which reduces the importance of communication times. On the contrary, *Payoff* may also favor a task with a large communication time, but only if the advance brought by its computation is really important (much more than what is required by *Advance* to schedule this task). Regarding *Johnson*, it achieves intermediate performance between *Payoff* and *Advance*. Its best variant (*Readiness*) achieves an average relative performance of 1.172. Johnson's algorithm [9], which is optimal without file sharing, does not adapt well to the general case, where tasks may share files.

A close observation of the results shows us that the differences between the heuristics are more significant when the communication-to-computation cost ratio is low. In the opposite case, it is likely that the communications from the master become the true bottleneck of all scheduling strategies.

As for the additional policies, *Readiness* brings a real gain. In comparison with the base heuristics, this policy enhances the performances by more than 8% on average, except for *Communication* for which the gain is negligible. *Readiness*, whose objective is to reduce the number of replicated files, reveals itself as specially effective. We can observe that, with *Communication*, tasks roughly depending on a same set of files tend to be put together in the task lists. So, when such a task is scheduled on some slave, it induces the transfer of the concerned files, and the other tasks which follow wind up at the beginning of the list. *Readiness* has thus little influence. This phenomenon is perfectly illustrated for the graphs of type Forks. For those graphs of type Forks, the difference between *Communication* and *Computation* is the most evident. Both heuristics perform badly in their basic versions, but *Computation+readiness* outperforms by 20% all the other heuristics, including the reference heuristics. It is because of these results for the Fork graphs that *Computation+readiness* ended up in Table 1, whereas *Communication+readiness* did not.

Locality does practically not change the performances of the heuristics. It seems that this policy, whose goal was to improve the locality with respect to the use of the files, is not aggressive enough. *Shared*, our last variant, does only affect significantly *Advance*, *Duration*, and *Johnson*: the former is improved (but does not become of good quality), while the last ones are degraded. With

Duration and *Johnson*, it is likely that *Shared* advantages tasks with large communication times, instead of those depending on highly shared files, as we wanted to. Another approach to better take file sharing into account, would be to reevaluate the ordering of the tasks on the processors, as files are being transferred.

In Table 1, we also report the computational costs of the heuristics (CPU time needed by each heuristic). The theoretical analysis is confirmed: our new heuristics are at least an order of magnitude faster than the reference heuristics.

As a conclusion, given their good performance compared to *Sufferage*, we believe that the eight new variants listed in Table 1 provide a very good alternative to the costly reference heuristics. The *Readiness* policy brings a large gain. As for the base heuristics, the simplest idea (among those that we evaluated) seems to work best: heuristics that only use an estimation of the execution time of the tasks on the processors. Depending upon the application graph, *Computation+readiness* and *Duration+readiness* are the recommended heuristics. The former performs better on Fork graphs, but the latter gives more stable results for all types of graphs.

6 Conclusion

In this paper, we have dealt with the problem of scheduling a large collection of independent tasks, that may share input files, onto heterogeneous clusters. On the theoretical side, we have shown new complexity results. On the practical side, we have improved upon the heuristics proposed by Casanova et al. [1,2]. We have succeeded in designing a collection of new heuristics which have similar performances but whose computational costs are an order of magnitude lower.

This work, as the one of Casanova et al., was limited to the master-slave paradigm. It is intended as a first step towards addressing the challenging situation where

- input files are distributed among several file servers (several masters) rather than being located on a single master,
- communication can take place *between* computational resources (slaves) in addition to the messages sent by the master(s): some slave may well propagate files to another slave while computing.

We hope that the ideas introduced when designing our heuristics will prove useful for this difficult scheduling problem. As shown by the preliminary results

reported in [13], much work remains to be done to design efficient mapping and scheduling strategies in a fully decentralized environment.

References

[1] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Heuristics for scheduling parameter sweep applications in Grid environments, in: Ninth Heterogeneous Computing Workshop, IEEE Computer Society Press, 2000, pp. 349–363.

[2] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Using simulation to evaluate scheduling heuristics for a class of applications in Grid environments, Research Report RR-1999-46, LIP, ENS Lyon, France (Sep. 1999).

[3] F. Berman, High-performance schedulers, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 1999, pp. 279–309.

[4] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, in: Eight Heterogeneous Computing Workshop, IEEE Computer Society Press, 1999, pp. 30–44.

[5] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, R. F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[6] B. A. Shirazi, A. R. Hurson, K. M. Kavi (Eds.), Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Computer Science Press, 1995.

[7] Ph. Chrétienne, E. G. Coffman, Jr., J. K. Lenstra, Z. Liu (Eds.), Scheduling Theory and its Applications, John Wiley and Sons, 1995.

[8] O. Beaumont, A. Legrand, Y. Robert, A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs, in: ISCIS XVII, Seventeenth International Symposium on Computer and Information Sciences, CRC Press, 2002, pp. 115–119.

[9] S. M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Research Logistics Quarterly 1 (1954) 61–68.

[10] E. M. Macambira, C. C. de Souza, The edge-weighted clique problem: valid inequalities, facets and polyhedral computations, European Journal of Operational Research 123 (2) (2000) 346–371.

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

[12] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files on heterogeneous clusters, Research Report RR-2003-28, LIP, ENS Lyon, France, also available as INRIA Research Report 4819 (May 2003).

[13] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files from distributed repositories, in: Euro-Par'04: Parallel Processing, Lecture Notes in Computer Science, Springer Verlag, 2004, pp. 148–159.