Scheduling tasks sharing files on heterogeneous master-slave platforms

Arnaud Giersch[1], Yves Robert[2], and Frédéric Vivien[2]

1: ICPS/LSIIT, UMR CNRS–ULP 7005, Strasbourg, France
2: LIP, UMR CNRS–ENS Lyon–INRIA–UCBL 5668
École normale supérieure de Lyon, France
`Arnaud.Giersch@icps.u-strasbg.fr, {Yves.Robert,Frederic.Vivien}@ens-lyon.fr`

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 [4, 3] although their cost is an order of magnitude lower.

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.

This paper is motivated by the work of Casanova, Legrand, Zagorodnov, and Berman [4, 3], who target the scheduling of tasks in APST, the AppLeS Parameter Sweep Template [2]. APST is a grid-based environment whose aim is to facilitate the mapping of application to heterogeneous platforms. Typically, an APST application consists of a *large* number of independent tasks, with possible input data sharing (see [4, 3] 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 tasks that depend upon the same files onto the same computational resource, so as to minimize communication requirements. Casanova et al. [4, 3] have considered three heuristics designed for completely independent tasks (no input file sharing) that were proposed in [9]. They have modified these three heuristics (originally called Min-min, Max-min, and Sufferage in [9]) 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 restrict to the same special case of the scheduling problem as Casanova et al. [4, 3]: 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 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 [4, 3] although their cost is an order of magnitude lower.

The rest of the paper is organized as follows. The next section (Section 2) is devoted 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 NP-completeness of the very specific instance of the problem where all files and tasks have the same size. 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.

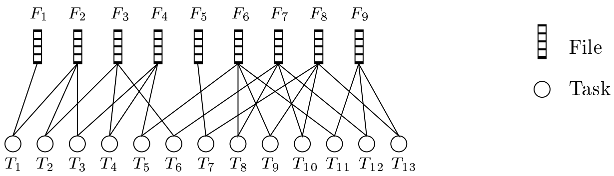


Figure 1. Bipartite graph gathering the relations between the files and the tasks.

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 there is an edge $e_{i,j} : F_i \to T_j$ 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 (in this example all tasks depend upon two files exactly; in the general case, each task depends upon an arbitrary number of files).

To summarize, the bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node in $V = \mathcal{F} \cup \mathcal{T}$ 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.

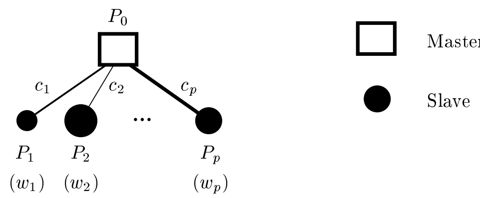


Figure 2. Heterogeneous fork-graph.

2.2. Platform graph

The tasks are scheduled and executed on a master-slave heterogeneous platform. We consider 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 let $\mathcal{P}$ denote the platform 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 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 five files (F_5 to F_9). In this schedule, files F_6 and F_7 must be sent to both slaves.

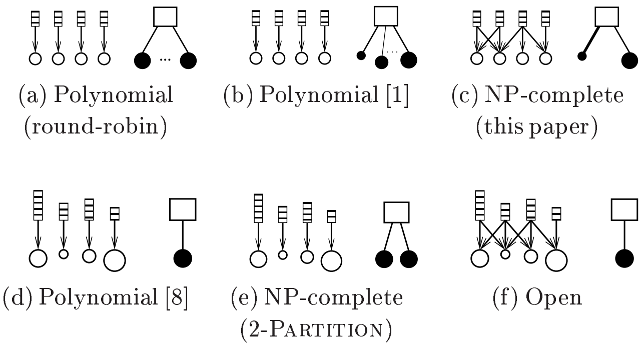
To summarize, we assume a fully heterogeneous master-slave paradigm: slaves have different speeds and links have different capacities. Communications from the master are serial, and may well become the major bottleneck.

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.

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



(a) Polynomial (round-robin) (b) Polynomial [1] (c) NP-complete (this paper)

(d) Polynomial [8] (e) NP-complete (2-Partition) (f) Open

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

3. Complexity

Most scheduling problems are known to be difficult [10, 5]. 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. 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.

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 [8] 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.

At the time of this writing, we do not know the complexity of the general instance with one slave (situation (f) of Figure 3). Because Johnson's algorithm is quite intricate, we conjecture that the decision problem associated to the general instance, where files are shared between tasks, is NP-complete. We do not even know what the complexity is when files are shared between tasks, but all tasks and files have the same size.

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 [1] for a description and proof.

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

Definition 1 (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 [6] 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 [6]).

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.

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

See the research report [7] for a proof.

4. Heuristics

In this section, we first recall the three heuristics used by Casanova et al. [4, 3]. 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. [4, 3], 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.

Structure of the heuristics. All the reference heuristics are built on the model presented in Figure 4.

```
S ← T        S is the set of the tasks that remain to be scheduled
while S ≠ ∅ do
  for each task T_j ∈ S and each processor P_i do
    Evaluate OBJECTIVE(T_j, P_i)
  Pick the "best" couple of a task T_j ∈ S and a processor P_i according to OBJECTIVE(T_j, P_i)
  Schedule T_j on P_i as soon as possible
  Remove T_j from S
```

Figure 4. Structure of reference heuristics.

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.

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:

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

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:

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

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 [4, 3] for details.

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: $O(p \cdot n^2 + p \cdot |\mathcal{E}|)$. 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 $O(p \cdot n^2 \cdot \log p + p \cdot |\mathcal{E}|)$.

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 scheduling 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 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 [8] 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 in Figure 5.

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

for any processor P_i **do**
  for any task $T_j \in \mathcal{T}$ **do**
    Evaluate OBJECTIVE(T_j, P_i)
  Build the list $L(P_i)$ of the tasks sorted according to the value of OBJECTIVE(T_j, P_i)
while there remain tasks to schedule **do**
  for any processor P_i **do**
    Let T_j be the first unscheduled task in $L(P_i)$
    Evaluate COMPLETIONTIME(T_j, P_i)
  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

Figure 5. Structure of the new heuristics.

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:

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

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:

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

The tasks are sorted by decreasing payoffs. Furthermore, the order of the tasks does not depend on the processor, so only one sorted list is required with this objective function. Note that the actual objective function to compute the payoff of scheduling task T_j on processor P_i would be: $\text{OBJECTIVE}(T_j, P_i) = \frac{t_j \cdot w_i}{\sum_{e_{k,j} \in \mathcal{E}} f_k \cdot c_i}$; 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.

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. 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 [1] 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: In the evaluation of the communication times performed for the objective functions, we replace the sum of the file sizes by the weighted sum of the file sizes divided by the number of tasks depending on these files. We obtain new objective functions which have the same name than the previous ones plus the tag "shared". 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 7 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 7 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 $L(P_i)$). This can be implemented with no overhead if each file is tagged with the processor (if any) it resides on.

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.

4.5. Computational complexity

Computing the value of an objective function for all tasks on all processors has a cost of $O(p \cdot (n + |\mathcal{E}|))$. So the construction of all the sorted lists has a cost of $O(p \cdot n \cdot \log n + p \cdot |\mathcal{E}|)$, except for the heuristics which only require a single sorted list and whose complexity is thus $O(n \cdot \log n + |\mathcal{E}|)$. The execution of the loop at Step 5 of the heuristics (see Figure 5) has an overall cost of $(p \cdot n \cdot |\mathcal{E}|)$. Hence the overall execution time of the heuristics is:

$$O(p \cdot n \cdot (\log n + |\mathcal{E}|))$$

We have replaced the term n^2 in the complexity of the reference heuristics by the term $n \cdot \log n$. 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.

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 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. Tasks graphs

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

Two-one: each task depends on exactly two files: one file which is shared with some other tasks, and one un-shared file.

Random: each task randomly depends on 1 up to 50 files.

Partitioned: this is a type of graph intermediate between the two previous ones; the graph is divided into 20 chunks of 100 tasks, and on each chunk each task randomly depends on 1 up to 10 files. The whole graph contains at least 20 different connected components.

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

Each of our graphs contains 2000 tasks and 2500 files, except for the fork graphs which also contain 2000 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 best ten heuristics, together with their cost. 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 in the 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.)

We see that **Sufferage** gives the best results: in average, it is within 11% of the optimal. The next nine heuristics closely follow: they are within 13% to 14.7% of the optimal. Out of these nine heuristics, only **Min-min** is a reference heuristic. Clearly, the readiness policy has a major impact on the results.

In Table 1, we also report computational costs (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.

We report more detailed performance data in [7].

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 differences between the heuristics are more significant when the communication-to-computation cost ratio is low. In the opposite case, is it likely that the communications from the master become the true bottleneck of all scheduling strategies. **Computation+readiness** outperforms by 20% all the other heuristics, including the reference heuristics, for the graphs of type Forks. However, the **Duration+readiness** gives more stable results for all types of graphs. Overall, **Computa-**

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	1.147	0.1234	1.618	0.4749
Max-min	1.504	0.4601	392.8	194.0

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

tion+readiness and **Duration+readiness** are the recommended heuristics.

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 a new complexity result. On the practical side, we have improved upon the heuristics proposed by Casanova et al. [4, 3]. 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.

References

[1] O. Beaumont, A. Legrand, and 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*, pages 115–119. CRC Press, 2002.

[2] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Research Report 99-46, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, Sept. 1999.

[4] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Ninth Heterogeneous Computing Workshop*, pages 349–363. IEEE Computer Society Press, 2000.

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

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

[7] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous clusters. Research Report RR-2003-28, LIP, ENS Lyon, France, May 2003.

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

[9] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Eight Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

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