Laboratoire de l'Informatique du Parallélisme

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

Scheduling tasks sharing files from distributed repositories

Arnaud Giersch,
Yves Robert,
Frédéric Vivien

October 2003

Research Report N° 2003-49

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

Scheduling tasks sharing files from distributed repositories

Arnaud Giersch, Yves Robert, Frédéric Vivien

October 2003

Abstract

This paper is devoted to scheduling a large collection of independent tasks onto a large distributed heterogeneous platform, which is composed of a set of servers. Each server is a processor cluster equipped with a file repository. The tasks to be scheduled depend upon (input) files which initially reside on the server repositories. A given file may well be shared by several tasks. For each task, the problem is to decide on which server to execute it, and to transfer the required files (those which the task depends upon) to that server repository. The objective is to find a task allocation, and to schedule the induced communications, 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, including an extension of the `min-min` heuristic to the decentralized framework, and several lower cost heuristics, which we compare through extensive simulations.

Keywords: Scheduling, heterogeneous clusters, grid, independent tasks, file-sharing, heuristics.

Résumé

Dans cet article, nous nous intéressons à l'ordonnancement d'un grand nombre de tâches indépendantes sur une plate-forme hétérogène distribuée composée d'un ensemble de serveurs. Chaque serveur est une grappe de processeurs doté d'un entrepôt de données. Les tâches à ordonnancer dépendent de fichiers (d'entrée) qui sont initialement stockés dans les entrepôts. Un fichier donné peut être partagé par plusieurs tâches. Pour chaque tâche, notre problème est de décider sur quel serveur l'exécuter, et de transférer les fichiers nécessaires (ceux dont dépend la tâche) vers l'entrepôt de ce serveur. L'objectif est de trouver une allocation des tâches, et un ordonnancement des communications induites, qui minimisent le temps total d'exécution. La contribution de cet article est double. Sur le plan théorique, nous établissons de nouveaux résultats de complexité qui caractérisent la difficulté du problème. Sur le plan pratique, nous proposons plusieurs nouvelles heuristiques, dont une extension de l'heuristique `min-min` aux plates-formes distribuées, et des heuristiques de moindre coût, que nous comparons grâce à des simulations.

Mots-clés: Ordonnancement, grappes hétérogènes, grilles de calcul, tâches indépendantes, partage de fichiers, heuristiques.

1 Introduction

In this paper, we are interested in scheduling independent tasks onto collections of 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. Initially, the files are distributed among several server repositories. Because of the computations, some files must be replicated and sent to other servers: before a task can be executed by a server, a copy of each file that the task depends upon must be made available on that server. For each task, we have to decide which server will execute it, and to orchestrate all the file transfers, so that the total execution time is kept minimum.

This paper is a follow-on of two series of work, by Casanova, Legrand, Zagorodnov, and Berman [3, 4] on one hand, and by Giersch, Robert, and Vivien [8, 9] on the other hand. In [3, 4], Casanova et al. target the scheduling of tasks in APST, the AppLeS Parameter Sweep Template [1]. 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. have considered three heuristics designed for completely independent tasks (no input file sharing) that were proposed in [10]. They have modified these three heuristics (originally called `min-min`, `max-min`, and `sufferage` in [10]) to adapt them to the additional constraint that input files are shared between tasks.

As 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 [8, 9], Giersch et al. have introduced several new heuristics, which are shown to perform as efficiently as the best heuristics in [3, 4] although their cost is an order of magnitude lower.

However, all the previous references restrict to a very special case of the scheduling problem: they 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.

In this paper, we deal with the most general instance of the scheduling problem: we assume a fully decentralized system, where several servers, with different computing capabilities, are linked through an interconnection network. To each server is associated a (local) data repository. Initially, the files are stored in one or several of these repositories (some files may be replicated). After having decided that server S_i will execute task T_j, the input files for T_j that are not already available in S_i local repository will be sent through the network. Several file transfers may occur in parallel along disjoint routes.

The contribution of this paper is twofold. On the theoretical side, we establish a complexity result that assesses the difficulty of the problem. On the practical side, we design several heuristics. The first heuristic is the extension of the `min-min` heuristic to the decentralized framework. This extension turns out to be surprisingly difficult, and we detail both the problems encountered, and the solution that was provided. The next heuristics aim at retaining the good performances of the `min-min` variants while reducing the computational cost by an order of magnitude.

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 TSFDR (*Tasks Sharing Files from Distributed Repositories*). Next, in Section 3, we establish a complexity result, namely the NP-completeness of the very specific instance of the problem where all files have the same size, all tasks have negligible (zero) cost, and all communication links have identical bandwidth. After this theoretical result, we move to the design of polynomial heuristics. In

Section 4, we start with the implementation of the `min-min` heuristic. We detail the difficulties linked to the routing (Section 4.2) and to the ordering of the communications (Section 4.3); then, in Section 4.4, we design an algorithm to schedule a set of communications whose target destination is the same server, a crucial step in the implementation of the `min-min` heuristic (Section 4.5). In Section 5 we deal with the design of low-cost polynomial-time heuristics to solve the TSFDR problem. We report some experimental data in Section 6. Finally, we state some concluding remarks in Section 7.

2 Framework

In this section, we formally state the optimization problem to be solved. We also work out a toy example.

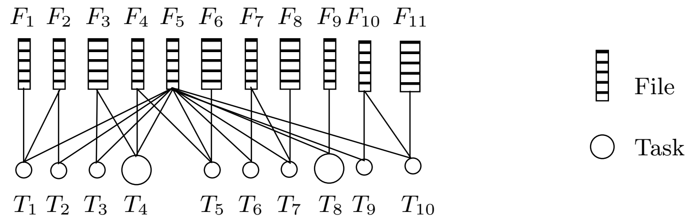

Figure 1: Bipartite graph gathering the relations between the files and the tasks.

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 (a task never takes as input the result of the computation of another task).

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 $e_{i,j} : F_i \rightarrow T_j$ in $\mathcal{E}$ if and only if task T_j depends on file F_i. Intuitively, files F_i such that $e_{i,j} \in \mathcal{E}$ contain data needed as input for the execution of task T_j. 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 $m = 11$ files and $n = 10$ tasks. For instance, task T_1 depends upon files F_1, F_2, and F_5, and file F_5 is an input to all tasks T_1 to T_{10}.

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.

2.2 Platform graph

The tasks are scheduled and executed on an heterogeneous platform composed of a set of *servers*, which are linked through a platform graph $\mathcal{P} = (\mathcal{S}, \mathcal{L})$. Each node in $\mathcal{S} = \{S_1, ..., S_s\}$ is a server, and each link $l_{i,j} \in \mathcal{L}$ represents a communication link from server S_i to server S_j. We assume that the graph $\mathcal{P}$ is connected, i.e. that there is a path between any server pair. By default, we assume that all links are bidirectional, hence $\mathcal{P}$ is undirected, but we could easily deal with oriented links.

Each server $S_i = (R_i, C_i)$ is composed of a local repository R_i, associated to a local computational cluster C_i. The files needed by the computations (the tasks) are stored in the repositories. We assume that a file may be duplicated, and thus simultaneously stored on several repositories.

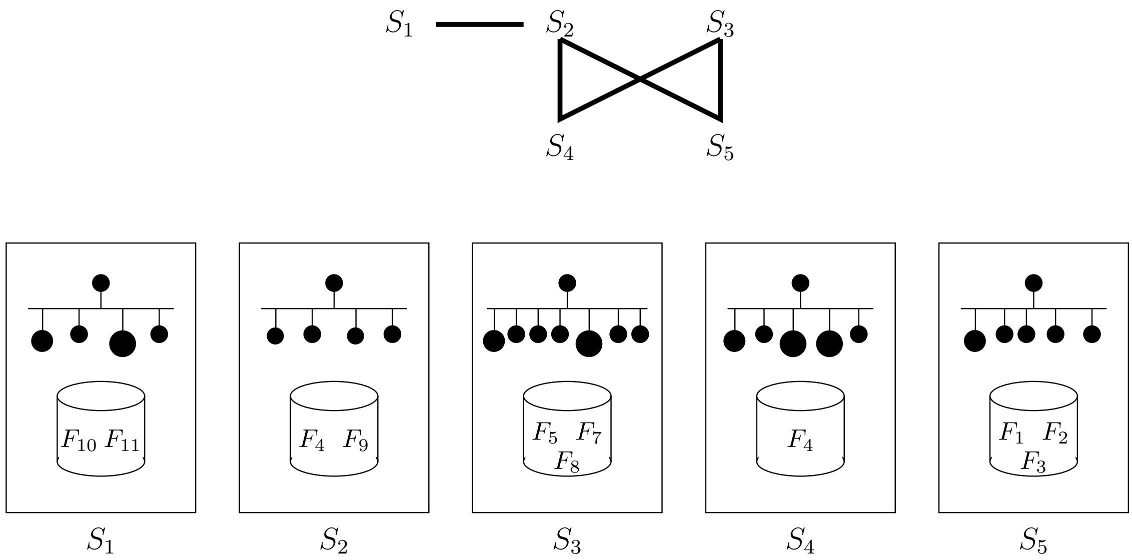

Figure 2: Platform graph, with the initial distribution of files to the server repositories.

We make no restriction on the possibility of duplicating the files, which means that each repository is large enough to hold a copy of all the files. See Figure 2 for an example of platform.

For a cluster to be able to process a task, the corresponding repository must contain all the files that the task depends upon. With the previous notations: for cluster C_i to be able to process task T_j, repository R_i must hold all files F_k such that $e_{k,j} \in \mathcal{E}$. Therefore, before C_i can start the execution of T_j, the server S_i must have received from the other server repositories all the files that T_j depends upon, and which were not already stored in R_i. For communications, we use the one-port model: at any given time-step, there are at most two communications involving a given server, one sent and the other received.

As for the cost of communications, consider first the case of adjacent servers in the platform graph. Suppose that server S_i sends the file F_j (stored in its repository R_i) to another server S_k, to which it is directly linked by the network link $l_{i,k} = l$. We denote by b_l the bandwidth of the link l, so that f_j/b_l time-units are required to send the file. Next, for communications involving distant servers, we use a store-and-forward model: we route the file from one server to the next one, leaving a copy of the file in the repository of each intermediate server. The cost is the sum of the costs of the adjacent communications. Leaving copies of transferred files on intermediate servers multiplies the number of potential sources for each file and is likely to accelerate the processing of the next tasks, hence the store-and-forward model seems quite well-suited to our problem.

Finally, we suppose that when the necessary files are on a server repository, they are available for free on its cluster. In other words, we assume no communication time between a cluster and its associated repository: the cost of intra-cluster messages is expected to be an order of magnitude lower than that of inter-cluster ones. We also assume that the only communication costs are due to the communication of files. Indeed, we consider no migration cost for assigning a task to a cluster. In another model, one could imagine that the code of a task originally lies on a repository and that it should also be sent to the repository linked to the cluster the task is assigned to. This model can easily be embedded in ours as one only need to add to our bipartite graph of relations between files and tasks some "virtual files" representing the task codes.

As for computation costs, each cluster C_i is composed of heterogeneous processors. More precisely, C_i gathers c_i processors $C_{i,k}$, $1 \leq k \leq c_i$. The speed of processor $C_{i,k}$ is $s_{i,k}$, meaning that $t_j/s_{i,k}$ time-units are needed to execute task T_j on $C_{i,k}$. A coarser approach is to view cluster C_i as a single computational resource of cumulative speed $\sum_{k=1}^{c_i} s_{i,k}$. We easily model the situation where a given server is composed of a single repository but has no computational capability: simply create a (fake) cluster of null speed.

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 processor of each cluster, and when. It must also decide the ordering in which the necessary files are sent from server repositories to server repositories. We stress three important points:

- Some files may well be sent several times, so that several clusters can independently process tasks that depend upon these files.
- A file sent to some repository remains available for the rest of the schedule; so, if two tasks depending on the same file are scheduled on the same cluster, the file must only be sent once.
- Initially, a file is available on one or several well-identified servers. But when routing a file from one of these servers to another one, a copy is left on each intermediate server repository. Hence all the intermediate servers, in addition to the server which was the final destination of the file in the communication, become potential sources for the file.

We let TSFDR($\mathcal{G}, \mathcal{P}$) (Tasks Sharing Files from Distributed Repositories) denote the optimization problem to be solved.

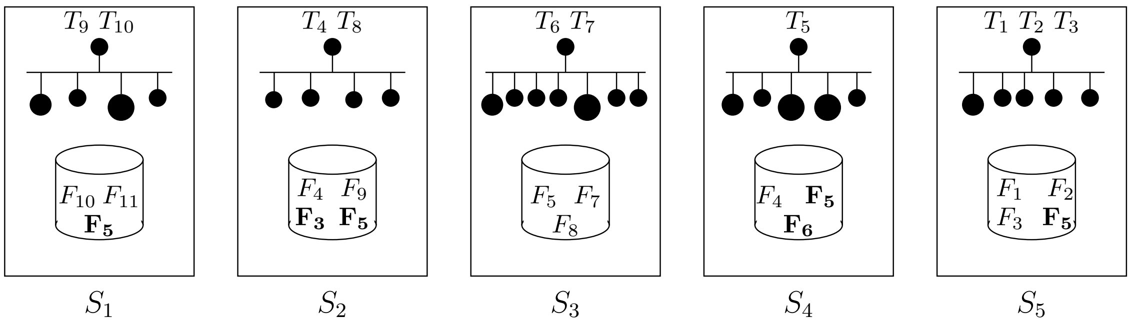

Figure 3: Platform graph, with task assigned to servers, and the final location of the files. Duplicated files are shown in bold.

2.4 Working out the example

Consider the example presented in Figures 1 (application graph) and 2 (platform graph). On Figure 2, we have also indicated the initial file distribution on the servers. Assume that the task mapping was decided to be the following, as illustrated on Figure 3:

- Server S_1 executes tasks T_9 and T_{10}
- Server S_2 executes tasks T_4 and T_8
- Server S_3 executes tasks T_6 and T_7
- Server S_4 executes task T_5
- Server S_5 executes tasks T_1, T_2, and T_3.

We do not discuss here how to determine such a mapping, although we point out that this is a difficult procedure, as shown later in the paper. Instead, we focus on the communications that are required by the mapping. Here is the list of the files needed by each server:

- Server S_1 needs files F_5 and F_{10} for T_9, and files F_5, F_{10} and F_{11} for T_{10}
- Server S_2 needs files F_3, F_4 and F_5 for T_4, and files F_5 and F_9 for T_8
- Server S_3 needs files F_5 and F_7 for T_6, and files F_5, F_7 and F_8 for T_7
- Server S_4 needs files F_4, F_5, and F_6 for T_5
- Server S_5 needs files F_1, F_2, and F_5 for T_1, files F_2 and F_5 for T_2, and files F_3 and F_5 for T_3.

Initially, S_1 holds F_{10} and F_{11} in its repository, so it needs to receive only file F_5. Note that S_1 needs to receive F_5 only once, even though it executes two tasks that depend upon this file. In fact, because each task depends upon F_5, each server except S_3 needs to receive F_5. In addition to F_5, S_2 needs to receive file F_3; S_3 does not need any extra file; S_4 needs F_4, F_5, and F_6, while S_5 only needs F_5.

Since S_3 is the only source for F_5, we have to decide an ordering for scheduling the transfers to the other processors. There are two routes from S_3 to S_2 in the platform, through S_4 or through S_5, and we have to decide which one to use. Also, we have to (try) to schedule independent computations in parallel: for instance the transfer of F_5 from S_3 to S_5 and the transfer of F_4 from S_2 to S_4 can take place in parallel. We come back to the different scenarios for routing and communication scheduling in Section 4.

3 Complexity

Most scheduling problems are known to be difficult [12, 5], and the TSFDR optimization problem is no exception. Heterogeneity may come from several sources: files or tasks can have different weights, while clusters or links can have different speeds. Simple versions of these weighted problems already are difficult. For instance the decision problem associated to the instance with no files and two single-processor clusters of equal speed already is NP-complete: in that case, TSFDR reduces to the scheduling of independent tasks on a two-processor machine, which itself reduces to the 2-Partition problem [7] as the tasks have different weights. Conversely, mapping equal-size files and equal-size tasks on a single server platform with two heterogeneous processors and two links of different bandwidths is NP-hard too [9].

The aim of this section is to prove that the simple fact of deciding where to move the files so as to execute the tasks is a difficult combinatorial problem, even in the un-weighted version where all files have same size and all communication links have same bandwidth. We even assume that all tasks have zero weight, or equivalently that all clusters have infinite speed. We then have a mapping problem: we have to map the tasks to the clusters and to gather each required set of files (that are input of a task) on a server repository, with the objective to minimize the number of communications steps. All communications have unit-time, but independent communications, i.e. involving distinct senders and distinct receivers, may well take place in parallel. Formally, we state the decision problem associated to this very particular instance of TSFDR as follows:

Definition 1 (TSFDR-Move-Dec($\mathcal{G}$, $\mathcal{P}$, K)). *Given a bipartite application graph $\mathcal{G} = (\mathcal{F} \cup \mathcal{T}, \mathcal{E})$, a platform graph $\mathcal{P} = (\mathcal{S}, \mathcal{L})$, assuming:*
- uniform file sizes ($f_i = 1$),
- homogeneous interconnection network ($b_i = 1$),
- zero processing time ($t_i = 0$ or $s_j = +\infty$),
and given a time bound K, is it possible to schedule all tasks within K time-steps?

Theorem 1. TSFDR-Move–Dec*($\mathcal{G}$, $\mathcal{P}$, K) is NP-complete.*

Proof. Obviously, TSFDR-Move-Dec($\mathcal{G}$, $\mathcal{P}$, K) belongs to NP. To prove its completeness, we use a reduction from the well-known MinCut problem, which is NP-complete [7]. More precisely we use a restriction of the MinCut problem where each vertex has a large degree, but we show that this restriction, which we denote as MinCutLargeDegree, remains NP-complete:

Definition 2 (MinCutLargeDegree($\mathcal{H}$,B)). *Given a non-oriented graph $\mathcal{H} = (V, E)$, with an even number of vertices, and a bound B, $1 \le B \le |V|$, and assuming that each vertex has a degree at least $B+1$, is there a partition $V = V_1 \cup V_2$ with $|V_1| = |V_2| = |V|/2$, such that the number of crossing edges does not exceed B: $|\{e = (u,v) \in E, u \in V_1, v \in V_2\}| \le B$?*

Lemma 1. MinCutLargeDegree*($\mathcal{H}$,B) is NP-complete.*

Proof. Obviously, MinCutLargeDegree($\mathcal{H}$,B) still belongs to NP. To prove its completeness, we use a reduction from the original MinCut problem: consider an arbitrary instance $\mathcal{I}_1$ of MinCut: given a graph $H' = (V', E')$, where $|V'|$ is even, and a bound B', is there a partition $V' = V_1' \cup V_2'$ with $|V_1'| = |V_2'| = |V'|/2$, such that $|\{e = (u,v) \in E' | u \in V_1', v \in V_2'\}| \le B'$? To construct an instance $\mathcal{I}_2$ of MinCutLargeDegree, we expend each vertex $v \in V'$ so as to form a "private" clique of size $B'+2$, so that each vertex has a degree at least $B'+1$. The edges of the cliques are the only ones added. Formally, $V = \{v_{i,k}, 1 \le i \le |V'|, 0 \le k \le B'+1\}$, and $E = \{(v_{i,0}, v_{j,0}) | (v_i, v_j) \in E'\} \cup \{(v_{i,k}, v_{i,k'}) | 0 \le k \ne k' \le B'+1, 1 \le i \le |V'|\}$. Intuitively, $v_{i,0}$ corresponds to the original vertex $v_i \in V'$ and $v_{i,j}$, $j \ge 1$, is a new vertex (in the clique). Finally, we let $B' = B$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$.

Assume first that $\mathcal{I}_1$ has a solution, i.e. an equal-size partition $V' = V_1' \cup V_2'$ with no more than B' crossing edges. We let $V_1 = \{v_{i,k} | 0 \le k \le B+1, v_i \in V_1\}$, and we let the other vertices in V_2. This leads to an equal-size partition of V. The number of crossing edges is the same, because each clique has been mapped on the same side of the partition, and there are no other edges involving new vertices. Hence a solution to $\mathcal{I}_2$.

Conversely, assume that $\mathcal{I}_2$ has a solution, i.e. an equal-size partition $V = V_1 \cup V_2$ with no more than B crossing edges. We claim that each clique has been mapped on the same side of the partition. Otherwise, for a given index i, we would have b vertices $v_{i,k} \in V_1$ and $B+2-b$ vertices $v_{i,k} \in V_2$, with $1 \le b \le B+1$. But there are $b(B+2-b) > B$ crossing edges linking these vertices, a contradiction. Now we easily derive the solution to $\mathcal{I}_1$: V_1' is the set of vertices v_i such that $v_{i,0} \in V_1$, and similarly for V_2. V_1 and V_2 are of equal size, and each clique is mapped on the same set, so V_1 and V_2 have the same number of cliques, hence V_1' and V_2' have same cardinal. Finally, all crossing edges are between cliques, hence their number is the same in the partition $V = V_1 \cup V_2$ as in the partition $V' = V_1' \cup V_2'$, hence the result. □

We now return to the proof of the completeness of TSFDR-Move–Dec($\mathcal{G}$,$\mathcal{P}$,K). We start from an arbitrary instance $\mathcal{I}_1$ of MinCutLargeDegree: given a graph $H = (V, E)$, where $|V|$ is even, and a bound B, $1 \le B \le |V|$, and assuming that each vertex has a degree at least $B+1$, is there a partition $V = V_1 \cup V_2$ with $|V_1| = |V_2| = |V|/2$, such that $|\{e = (u,v) \in E | u \in V_1, v \in V_2\}| \le B$? Let $|V| = 2p$, and (without loss of generality) assume that $p \ge 3$ and $B \le p-1$. See Figure 4 for an illustration with $p = 4$ and $B = 2$.

We construct the following instance $\mathcal{I}_2$ of TSF2DR-Move-Dec($\mathcal{G}$,$\mathcal{P}$,K). To each $v_i \in V$ we associate a file $f_i \in \mathcal{F}$. We also introduce a collection of new files:

- $(p+1)B$ files $x_{i,j}$, $1 \le i \le B, 1 \le j \le p+1$,
- $(p-2)B$ files $y_{i,j}$, $1 \le i \le B, 1 \le j \le p-2$,
- and $p+1$ files z_i, $1 \le i \le p+1$,

so that there is a total of $m = 3p+1+B(2p-1)$ files in $\mathcal{F}$.

As for tasks, there are as many tasks as edges in the original graph H, plus B additional tasks, so that $\mathcal{T}$ comprises $n = |E| + B$ tasks. More specifically, we create a task $T_{i,j}$ for each edge $e = (v_i, v_j) \in E$, and we add B tasks denoted as T_i', $1 \le i \le B$. The relations between tasks and files are defined as follows. First, if $(v_i, v_j) \in E$, there are two edges in $\mathcal{E}$, one from file f_i to task $T_{i,j}$, and the other from f_j to $T_{i,j}$. The files f_i and f_j are not the only inputs of task $T_{i,j}$, which also depends upon all files z_k. In other words, we add $(p+1)|E|$ edges $(z_k, T_{i,j})$ in $\mathcal{E}$. Then, the last B tasks T_i' all have $2p-1$ input files: given i, there is an edge from each $x_{i,j}$ and from each $y_{i,j}$ to T_i'. In summary, there are $(p+3)|E| + (2p-1)B$ edges in $\mathcal{E}$. See Figure 5 for an illustration.

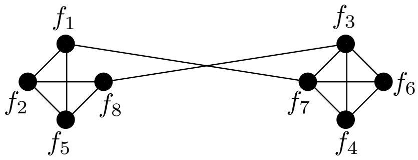

Figure 4: The original graph in the MinCutLargeDegree instance.

There remains to describe the platform graph $\mathcal{P}$. There are $2p+2B+2$ servers in $\mathcal{P}$, which we denote F_i, for $1 \leq i \leq 2p$, X_i and Y_i, for $1 \leq i \leq B$, and W_1 and W_2. There is a communication link from each F_i to W_1, to W_2, and to each X_j, which amounts to $2p(B+2)$ links. There are B additional links, namely from Y_i to X_i, hence a total of $2p(B+2)+B$ links in $\mathcal{P}$. See Figure 6 for an illustration. The initial distribution of files to server repositories is the following:

- file f_i is stored in F_i, for $1 \leq i \leq 2p$,
- files $x_{i,j}$, $1 \leq j \leq p+1$, are stored in X_i, for $1 \leq i \leq B$,
- files $y_{i,j}$, $1 \leq j \leq p-2$, are stored in Y_i, for $1 \leq i \leq B$,
- and files z_k, $1 \leq k \leq p+1$, are duplicated $B+2$ times, to be stored in W_1, in W_2, and in each X_i, $1 \leq i \leq B$.

Finally, we let $K = p$ for the scheduling bound. This completes the description of $\mathcal{I}_2$, whose size is clearly polynomial in the size of $\mathcal{I}_1$. As specified in the problem, all files have unit size, all communication links have unit bandwidth, and computation is infinitely fast.

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 there is a partition $V = V_1 \cup V_2$ with $|V_1| = |V_2| = p$, such that $b = |\{e = (u,v) \in E | u \in V_1, v \in V_2\}| \leq B$. We decide to compute task T'_i in server X_i. For each edge $(v_i, v_j) \in E$ such that both v_i and v_j belong to V_1, we execute task $T_{i,j}$ in server W_1. We make a similar decision if both v_i and v_j belong to V_2, then executing the task in W_2. There remains b tasks to allocate, those corresponding to crossing edges. Since $b \leq B$, we assign one of these tasks to each of the first X_i servers, $1 \leq i \leq b$. This allocation implies $B(p-2)+2p+2b$ communications, namely $p-2$ for each task T'_i, one for each file f_i to either W_1 or W_2, and two for each crossing edge. We have to orchestrate these communications within $K = p$ time-steps, without violating any one-port constraint. Intuitively, W_1 receives p files f_i, where $v_i \in V_1$, and similarly for W_2: these $2p$ messages are independent and can be scheduled at any time-step. Each X_i receives $p-2$ files $y_{i,j}$ from Y_i, so it has two time-slots left, exactly what is needed to receive the two files needed to execute one crossing edge task. Each server F_i sends its file f_i at most $B+1$ times, one time to one of the two W servers, and as many times as v_i is an incident vertex to a crossing edge. Rather than describing the schedule by extension, which would be cumbersome, we resort to the big artillery: we build a bipartite graph with senders on one side (the F and Y servers), receivers on the other side (the X and W servers), and we insert one edge for each communication to be scheduled. As $B+1 \leq p$, the maximal degree of each node in the graph does not exceed p. König's edge coloring theorem [11, chapter 20] ensures that we can partition all the edges into p disjoint matchings, and we implement one matching (which by definition involves independent communications) per time-step.

Overall, we obtain a valid scheduling to execute all tasks within p time-steps, hence a solution to $\mathcal{I}_2$.

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

1. Necessarily, server X_i executes task T'_i. Otherwise, if another server would execute it, the $p+1$ files $x_{i,j}$ (for all j) which reside in (the repository of) X_i should be transferred to the other server; since X_i is the only source for these files, this would require $p+1$ outgoing messages from X_i, which is impossible within p steps.

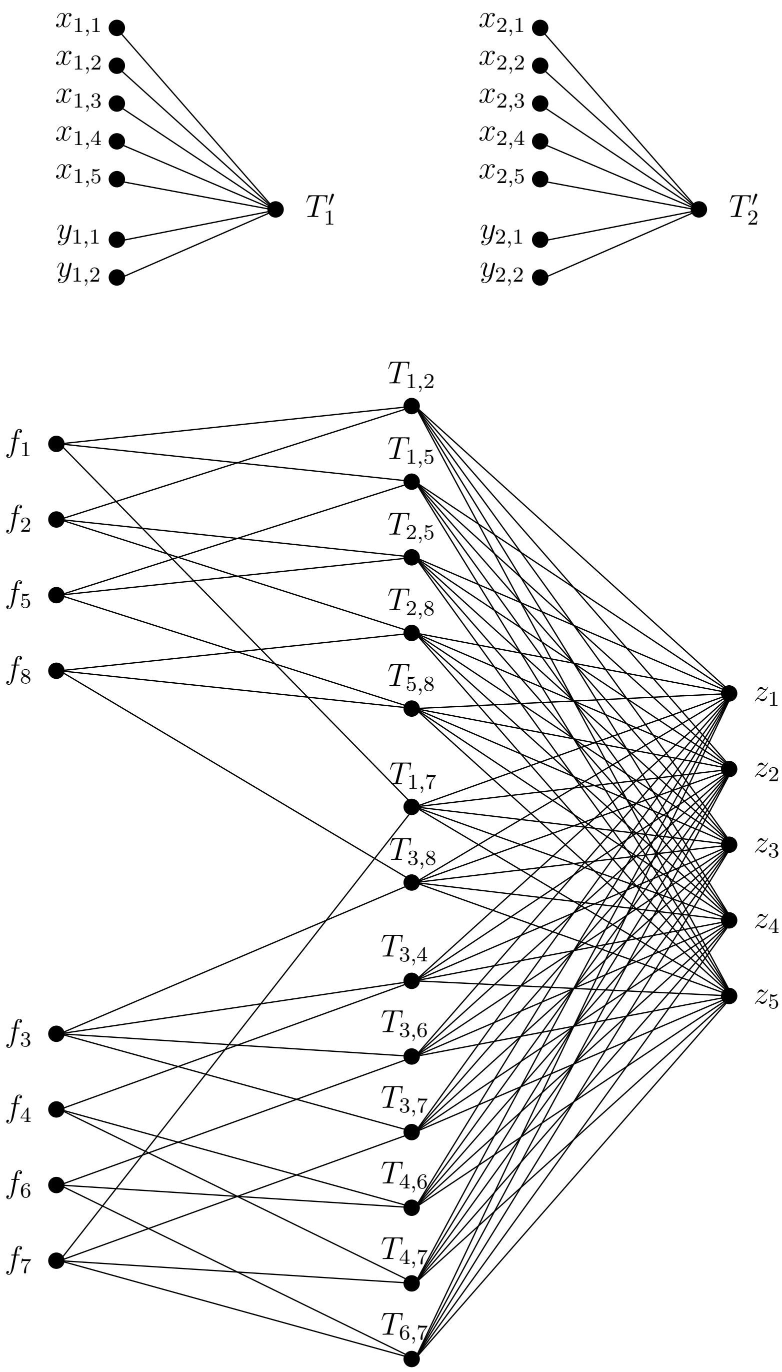

Figure 5: The bipartite application graph used in the reduction.

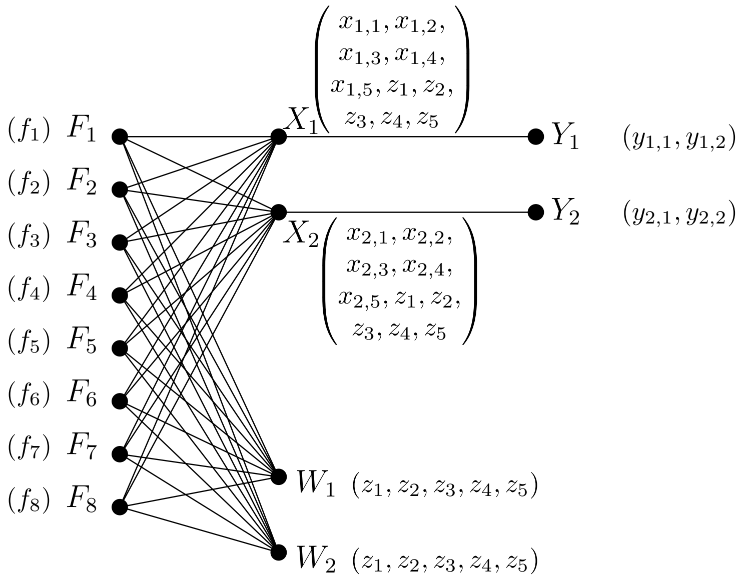

Figure 6: The platform graph used in the reduction.

2. With the same reasoning, a server labeled F or Y cannot execute any of the tasks $T_{i,j}$, because these tasks depend upon the $p+1$ files z_k whose sources lie in the servers labeled X, and in W_1 and W_2.

3. Because X_i executes T'_i, it must receive the $p-2$ files $y_{i,j}$ (for all j), whose only source is server Y_i. Hence X_i can receive at most two other files during the p steps of the schedule. As a consequence, X_i can execute at most one task $T_{j,k}$, if it indeed receive the two missing files.

4. How many files f_i have been received by W_1 and W_2? Let V_1 denotes the set of the files f_i sent by the F servers to W_1, and V_2 those sent to W_2. We do not know yet whether some files have been sent to both W_1 and W_2. But assume for a while that a given file f_i has neither been sent to W_1 nor to W_2. All the tasks $T_{i,j}$, where $(i,j) \in E$, must have been executed elsewhere. The only candidates are servers labeled X. But there are B of them, and each can execute at most one such task, while the degree of v_i in H is at least $B+1$, a contradiction.

5. Therefore, each file f_i has been sent either to W_1 or to W_2 (or both). But there is a total of $2p$ files to send to two servers within p time-steps: each server can receive at most p files. So no file has been sent twice. We conclude that V_1 and V_2 form a partition of V, with $|V_1| = |V_2| = p$.

6. There remains to show that there are no more than B edges crossing the partition $V = V_1 \cup V_2$. But these crossing edges correspond to tasks that can only be executed by the X servers, hence the result.

Finally, we have characterized the solution to $\mathcal{I}_1$. □

4 Extending the `min-min` scheme to distributed repositories

As the TSFDR scheduling problem is NP-complete, we look for polynomial heuristics to solve it. Considering the work of Casanova et al. [3, 4] for master-slave systems with a single server, we start by adapting the `min-min` scheme. During this study, we will justify some of our assumptions and simplifications concerning the communication model. Later, in Section 5, we introduce several new heuristics, whose main characteristic is a lower computational complexity than that of the `min-min` scheme.

This section is organized as follows. We recall the basic principle of the `min-min` heuristic in Section 4.1. We detail the difficulties linked to the routing in Section 4.2, and those related to the ordering of the communications in Section 4.3. Then, in Section 4.4, we design an algorithm to schedule a set of communications whose target destination is the same server, a crucial step when allocating a new task to a server in the implementation. We outline the final design of the `min-min` heuristic in Section 4.5.

4.1 Principle of the `min-min` scheme

The principle of the `min-min` scheme is quite simple:

- While there remain tasks to be scheduled do
  1. for each processor $C_{i,j}$ in the system and each task T_k that remains to be scheduled, evaluate the minimum completion time of T_k if mapped on $C_{i,j}$;
  2. pick a couple $(C_{i,j}, T_k)$ whose minimum completion time is minimal, and schedule the task T_k on the processor $C_{i,j}$.

The problem with this heuristic is to evaluate the Minimum Completion Time (MCT). When trying to schedule a task on a given processor, one has to take into account which required files already reside in the corresponding repository, and which should be routed through the network of servers. It is straightforward to determine which communications should take place. However, the scheduling of these communications is NP-complete in the general case, as discussed below.

Once we have decided how to schedule the communications, there only remains to evaluate the computation time. We heuristically decide to view a whole cluster as a single processor whose processing power is the sum of the processing powers of the cluster processors (the coarse-grain approach alluded to in Section 2.2). The start date for the execution of the new task is set as the latest date between (1) the arrival of the last of the necessary files and (2) the end of the (coarse evaluation of the) computation of the tasks assigned so far to the cluster. With this simplified coarse-grain view for computations, the only problem we are left with is the scheduling of the communications.

4.2 Complexity of scheduling communications with free routing

We deal with the situation where all communications have the same destination (namely the server that will execute the task). In the 1-port model, if the routing in the platform graph is not fixed, then this simple scheduling problem already is NP-hard. We first formally define the decision problem and then prove its NP-completeness:

Definition 3. CommSchedFreeRoute*$(\mathcal{P}, \mathcal{M}, D, T)$: Given a platform graph $\mathcal{P} = (\mathcal{S}, \mathcal{L})$, a finite set $\mathcal{M}$ of communications of same destination D, and a time bound T, is there a valid scheduling of the communications whose makespan is not greater than T? Each element of $\mathcal{M}$ is a couple (S_i, s) representing the communication of a file of size s, from the server S_i to the destination server D.*

Theorem 2. CommSchedFreeRoute*$(\mathcal{P}, \mathcal{M}, D, T)$ is NP-complete.*

Proof. Obviously, CommSchedFreeRoute$(\mathcal{P}, \mathcal{M}, D, T)$ belongs to NP. To prove its completeness, we use a reduction from 2-Partition, which is NP-complete [7]. Consider an arbitrary instance $\{a_1, a_2, ..., a_n\}$ of 2-Partition, where the a_i are strictly positive integers: is there a subset I of $\{1, ..., n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

From this instance of 2-Partition, we build the platform $\mathcal{P} = (\mathcal{S}, \mathcal{L})$ represented in Figure 7. In this platform, there are $n + 5$ servers:

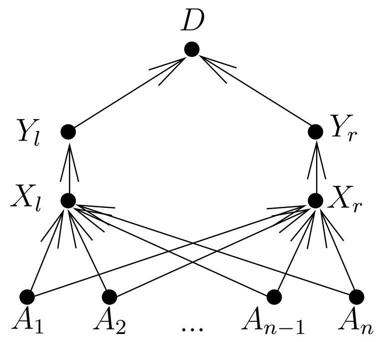

Figure 7: Graph used in the reduction from 2-PARTITION to COMMSCHEDFREEROUTE.

- $A_1, ..., A_n$: the server A_i stores a file F_i of size a_i.
- X_l and X_r: from each of the n servers $A_1, ..., A_n$, there is a link of bandwidth 1 to the server X_l and and one to the server X_r.
- Y_l and Y_r: there is a link of bandwidth $\frac{1}{2N}$ from the server X_l to the server Y_l, and one from X_r to Y_r, where $N = \sum_{1 \le i \le n} a_i$.
- D: there is a link of bandwidth 1 from the servers Y_l and Y_r to the server D.

The set, $\mathcal{M}$, of communications is defined by communications from the A servers to D: $\mathcal{M} = \{(A_i, a_i)\}_{1 \le i \le n}$. The time bound is $T = N^2 + N + \min_{1 \le i \le n} a_i$. Clearly the size of the constructed instance of COMMSCHEDFREEROUTE is polynomial in the size of the original instance of 2-PARTITION.

Assume that the original instance of 2-PARTITION admits a solution: let $(\mathcal{I}_1, \mathcal{I}_2)$ be a partition of $\{1, ..., n\}$ such that $\sum_{i \in \mathcal{I}_1} a_i = \sum_{i \in \mathcal{I}_2} a_i = \frac{N}{2}$. Without loss of generality, suppose that $\mathcal{I}_1$ contains an element a_j which is minimal: $a_j = \min_{1 \le i \le n} a_i$. We derive a scheduling for the instance COMMSCHEDFREEROUTE as follows:

1. At date 0, A_j sends its file of size a_j to X_l, which is received at date a_j.
2. Between dates 0 and $\frac{N}{2}$, the servers corresponding to $\mathcal{I}_2$ ($\{A_i\}_{i \in \mathcal{I}_2}$) send their files, in any order, to X_r.
3. Between dates a_j and $a_j + 2Na_j$, X_l sends the file received from A_j to Y_l.
4. Between dates a_j and $\frac{N}{2}$, the servers $\{A_i\}_{i \in \mathcal{I}_1, i \ne j}$ send their files, in any order, to X_l.
5. From $a_j + 2Na_j$ to $a_j + 2N\frac{N}{2}$, X_l sends to Y_l, in any order, the files received from the servers in $\{A_i\}_{i \in \mathcal{I}_1, i \ne j}$.
6. Between dates $\frac{N}{2}$ and $\frac{N}{2} + 2N\frac{N}{2}$, X_r sends to Y_r, in any order, the files received from the servers corresponding in $\mathcal{I}_2$.
7. From $a_j + 2N\frac{N}{2}$ to $a_j + 2N\frac{N}{2} + \frac{N}{2}$, Y_l sends, in any order, all the files that it holds to D.
8. From $a_j + 2N\frac{N}{2} + \frac{N}{2}$ to $a_j + 2N\frac{N}{2} + N$, Y_r sends, in any order, all the files that it holds to D.

Therefore, we have derived a valid scheduling that matches the time bound, hence a solution to the COMMSCHEDFREEROUTE instance.

Reciprocally, assume that the COMMSCHEDFREEROUTE instance admits a solution, hence a valid scheduling satisfying the time bound T. Then, the communication originating from any A_i must go either through X_l or X_r. We define as $\mathcal{I}_1$ (respectively $\mathcal{I}_2$) the A_i servers whose files are sent to D through X_l (resp. X_r). Thus, $\mathcal{I}_1$ and $\mathcal{I}_2$ defines a partition of $\{1, ..., n\}$. Suppose this is not a solution to 2-Partition. Then, without loss of generality, $\sum_{i\in\mathcal{I}_1} a_i > \sum_{i\in\mathcal{I}_2} a_i$ and thus $\sum_{i\in\mathcal{I}_1} a_i > \frac{N}{2}$. Thus, $\sum_{i\in\mathcal{I}_1} a_i \geq 1 + \frac{N}{2}$. Then, the time needed by X_l to send to Y_l all the files it received from the A_i's is equal to $2N \cdot (\sum_{i\in\mathcal{I}_1} a_i) \geq 2N\frac{N}{2} + 2N > min_{1\leq i\leq n} a_i + 2N\frac{N}{2} + N = T$, which is absurd. □

Theorem 2 shows that scheduling the communications is NP-complete as soon as we have the freedom to chose the route followed by the files. Therefore we assume that the routing from one server to another is fixed. Note that this is a realistic assumption in practice, because the routing is usually decided by table lookup. Between each pair of servers, if multiple routes are available, we decide to chose one among all the routes of maximal bandwidth. Furthermore, we decide the whole routing to be coherent: if S is the first server in the route from a server R to a server T, then the route from R to T is made of the physical link between R and S and of the route from S to T.

4.3 Complexity of communication scheduling with respect to an history

When trying to schedule the communications required for a task T_k, one must take into account the communications scheduled for the tasks previously scheduled. In other words, when trying to schedule the communications for a new task, one must take into account that the communication links are already used at certain time slots due to previously scheduled communications. Even with our hypothesis of a fixed routing, this leads to an NP-complete problem, as we now show:

Definition 4. CommSchedWithHistory*($\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T$): Given a platform graph $\mathcal{P} = (\mathcal{S}, \mathcal{L})$, a finite set $\mathcal{M}$ of communications of same destination D, an history $\mathcal{H}$ specifying for any communication link at which time-slots it is unavailable due to previously scheduled communications, and a time bound T, is there a valid scheduling of the communications whose makespan is not greater than T?*

Theorem 3. *Problem* CommSchedWithHistory*($\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T$) is NP-complete.*

Proof. Obviously, CommSchedWithHistory($\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T$) belongs to NP. To prove its completeness, we use a straightforward reduction from 2-Partition. Consider an arbitrary instance $\{a_1, a_2, ..., a_n\}$ of 2-Partition with $\sum_{1\leq i\leq n} a_i = N$. From this instance, we build a simple platform composed of two servers linked by a single communication link which is available at any time t in $[0..\frac{N}{2}] \cup [1 + \frac{N}{2}.. + \infty[$. We let $T = N + 1$. Then, one can easily see that there is a solution for the instance 2-Partition if and only if there is a solution to the corresponding instance of CommSchedWithHistory. To reach this result, we make the assumption that communications are never preempted, i.e. that a file is sent in one time and is not cut in several sub-pieces which would be sent at different non contiguous times. □

As a consequence of this result, we (heuristically) decide not to interleave previously scheduled communications with next communications to be scheduled. Therefore, a new communication will always be scheduled *after* any communication already scheduled on the same link.

4.4 Communication scheduling algorithm

Because of the two complexity results above, we have decided so far:

1. that the routing of communications is fixed;

2. that new communications are always scheduled after communications using the same links.

Under these hypotheses, we are able to design a greedy algorithm to schedule the communications required to send the necessary files to the server which we want to assign the new task to. In the following, we call

- *transfer* the communication, through the network of servers, of a file from the source server (which stores the file in its repository) to the destination server (where we attempt to schedule the new task);
- *local communications* the communications between neighbor servers; therefore, a transfer is potentially made up of several local communications.

In order (i) to avoid contentions and (ii) to enforce the one-port model, we would *a priori* need to memorize the availability dates of the links, in addition to the availability dates, in emission and in reception, of the servers. In fact, we do not need to store any data for the links: for each server S, its availability date in emission, denoted as $d_{out}(S)$, is equal to the maximum of the availability date of its outgoing communication links. The same holds true for its availability date $d_{in}(S)$ in reception. In addition to the availability dates of the servers, we also need to store the arrival date of a file on an intermediate server (between the source and the destination of a transfer).

Algorithm 1 presents our greedy algorithm: on each intermediate server we schedule the local communications as soon as possible, the intermediate servers being considered in decreasing distance from the destination of transfers.

Algorithm 1 Schedule a set of transfers of same destination D

from the platform graph $\mathcal{P}$, extract the tree $\mathcal{T}$ of the transfers to D.
for each server S in $\mathcal{T}$ **do**
  compute the depth of S (starting from 0 for D)
for each server S source of a transfer of a file F_i **do**
  add to the set of communications to be scheduled, the sending of file F_i from S to father(S) with start date $d_{out}(S)$
for $h = \text{height}(\mathcal{T}) - 1$ downto 0 **do**
  for each server S of depth h **do**
    while there remains to schedule some local communications **do**
      take any local communication c of file F_i from R to S of earliest start date d
      $start \leftarrow \max\{d, d_{out}(R), d_{in}(S)\}$
      $end \leftarrow start + f_i/b_{R\to S}$
      schedule c between dates *start* and *end*
      $d_{out}(R) \leftarrow end$
      $d_{in}(S) \leftarrow end$
      if $S \neq D$ **then**
        add to the set of communications to be scheduled, the sending of file F_i from S to father(S) with start date *end*

Of course, the algorithm does not always compute the optimal solution. The general problem is most likely NP-complete, due to its similarity with the flow-shop problem [7]. Here is a simple example where the algorithm fails to return the optimal solution. Consider two files F_1 and F_2, of sizes $f_1 = 2$ and $f_2 = 4$, to be routed from server A to server D via server B. The bandwidth of the link from A to B is 1, that of the link from B to D is 0.5. The availability dates of F_1 in server A is $t = 1$ while that of F_2 is $t = 0$. The greedy algorithm will route F_2 before F_1 on each link, starting at $t = 0$: the sending of F_2 terminates at $t = 4$ in B and at $t = 12$ in D; F_1 follows at $t = 6$ in B and $t = 16$ in D. However, routing F_1 first is better, even though we cannot start before $t = 1$: F_1 reaches B at $t = 3$ and D at $t = 7$, while F_2 reaches B at $t = 7$ and D at $t = 15$.

Complexity

If we denote by n_t the number of transfers and by h the height of the communication tree, the complexity of the algorithm is $O(n_t \cdot h + n_t \log n_t)$. Indeed, one does need to sort the file availability dates on the tree leaves. However, each intermediate servers receives sorted lists from each of its sons in the tree, and merging these lists costs at most $O(n_t)$ on each on the $h-2$ heights concerned.

More specifically, if we denote by

- ΔT the maximum degree of a task, i.e. the maximum number of files that a task depends upon;
- $\Delta\mathcal{P}$ the diameter of the platform graph;

the complexity of this algorithm is in $O(\Delta T \cdot \Delta\mathcal{P} + \Delta T \log(\Delta T))$.

4.5 Outline of the whole `min-min` scheme

Algorithm 2 presents our implementation of the `min-min` scheme on distributed repositories. The complexity of the whole heuristic is

$$O(n^2 s(\Delta T \cdot \Delta\mathcal{P} + \Delta T \log(\Delta T)) + n \max_{1 \leq i \leq s} c_i).$$

The last term comes from scheduling the tasks on the clusters. Indeed, once the communications are scheduled, we have for each task the availability date of the files which it depends upon at a cost $O(|\mathcal{E}|)$. On each cluster, we greedily schedule the tasks on the available processors. Among the tasks of lowest availability date, we take the largest one and schedule it on the processor on which it will have the minimum completion time (taking into account the date at which this processor will be available, knowing which tasks were already scheduled on it). The complexity of this task scheduling is $O(n \max_{1 \leq i \leq s} c_i)$.

We remark that going from systems with a single repository, as in [9], to systems with several repositories, the complexity increases by a factor $(\Delta T \cdot \Delta\mathcal{P} + \Delta T \log(\Delta T))$, corresponding to the cost of the communication scheduling algorithm. In the meantime, the number of processors was replaced by the number of servers. This is because of our simplified view of the problem: in the decision phase, we see each cluster C_j as a single computational resource of cumulative speed $\sum_{k=1}^{c_j} s_{j,k}$.

Algorithm 2 Outline of the whole `min-min` scheme

while there remains tasks to be scheduled **do**
  for each remaining task T_i **do**
    for each server S_j **do**
      use Algorithm 1 to compute the date t at which all the files required for the execution of T_i will be available on S_j
      evaluate the minimum completion time of T_i on S_j knowing t (as computed above) and considering the cluster C_j as a single processor
  pick a couple (S_j, T_i) whose minimum completion time is minimal
  map T_i on the cluster C_j
  use Algorithm 1 to schedule the communications needed by the execution of T_i on S_j
  schedule task T_i on the processor $C_{j,k}$, $1 \leq k \leq c_k$, which provides the minimum completion time

4.6 Variant of the `min-min` scheme

Our framework promotes file duplication. When a given server S requires a file F for the execution of some task T, the file may be available on several servers. A static policy would be to transfer the file from the server originally storing it, or more precisely from the closest server (in terms of communication time) that originally stored it, in the case of an initial duplication of the file. This naive scheme has the advantages of its simplicity: it is cheap and does not require to memorize file transfers. A more dynamic policy would be to choose, as the source of transfer, the closest server among the ones holding the file at the time where we consider the communication. We call this policy *closest*. Hopefully, such a policy can decrease communication costs. But it requires to dynamically update a list of possible sources for each file. Each time the *closest* policy is used, it induces a complexity increase of $O(s\Delta T)$, hence an overall complexity increase of $O(n^2s^2\Delta T)$ in the heuristic.

Of course, a third policy would be to invoke Algorithm 1 on all possible sets of transfer sources and to choose the best solution. Knowing the complexity of `min-min` heuristics, such an expensive policy is not realistic.

In the rest of this paper, we denote by `min-min` the heuristic following the first policy, and by `min-min+closest` the heuristic following the second policy.

5 Heuristics of lower complexity

As appealing as the `min-min` scheme could be because of the quality of the scheduling that it produces [3, 9], its computational cost is huge and may forbid its use. Therefore, we aim at designing heuristics which are an order of magnitude faster, while trying to preserve the quality of the scheduling produced.

Our heuristics are based on a *cost* function which tries to estimate the completion time of the execution a given task on a given server. We have designed two types of heuristics: static ones where costs are estimated once and for all; and dynamic ones where costs are reevaluated as mapping and scheduling decisions are being made.

5.1 Static heuristics

Algorithm 3 presents the structure of our first static heuristic. Step by step, we describe its different parts, and we explain our design choices.

Algorithm 3 Structure of the `static1` heuristic

```
for each task T_i do
    for each server S_j do
        compute the cost a(T_i, S_j)
    map T_i on a server S(T_i) such that a(T_i, S(T_i)) = min_{1≤j≤s} a(T_i, S_j)
for each task T_i, in increasing order of the costs a(T_i, S(T_i)) do
    use Algorithm 1 to schedule the communications needed by the execution of T_i on S(T_i)
for each server S_j do
    greedily schedule the tasks mapped on S_j
```

The cost function

In our fully static heuristics, the cost of a task T_i on a server S_j is defined as an evaluation of the minimum completion time of T_i on S_j. Following what has been previously done for the `min-min` heuristic, the minimum completion time is defined as the sum of the time needed to send the required files to S_j, as computed by Algorithm 1, plus the time needed by the cluster C_j to process the task, when the cluster is seen as a single computational resource:

$$a(T_i, S_j) = \text{Time-Algo-1}\,\{\mathit{comm}(\mathit{source}(F_k, S_j), S_j, F_k) \mid e_{k,i} \in \mathcal{E}, F_k \notin R_j\} + \frac{t_i}{\sum_{1\leq l\leq c_j} s_{j,l}} \quad (1)$$

where

- $\mathit{source}(F_k, S_j)$ denotes the server which initially stores F_k; in fact, since there may be several sources for F_k, we use the one closest to S_j (in terms of communication time);
- $\mathit{comm}(S, S', F)$ denotes the time to transfer file F from S to S'.

The overall complexity of the evaluation of the costs is $O(ns\Delta T \cdot \Delta \mathcal{P})$. We even speed-up these evaluations by approximating the time needed to send all the necessary files. Instead of using the precise but costly Algorithm 1, we use the simple formula:

$$Op_{e_{k,i}\in\mathcal{E}, F_k\notin R_j}\, comm(source(F_k, S_j), S_j, F_k), \tag{2}$$

where Op can design either the "sum" operator (over-approximation by sequentialization of all the communications) or the "max" operator (under-approximation by considering all the communications to take place in parallel). If we precompute once and for all the cost of sending an elementary file between any two servers, which requires $O(s^3)$ operations, the overall complexity of the evaluation of the costs drops to $O(n|\mathcal{E}| + s^3)$ with our approximations.

Task scheduling

To schedule the communications, we use Algorithm 1, the source of the file transfers being chosen according to the *closest* policy defined in section 4.6.

Once the communications are scheduled, we have for each task the availability date of the files which it depends upon at a cost $O(|\mathcal{E}|)$, and we proceed as explained in Section 4.5 for the `min-min` heuristic, with a cost $O(n \max_{1\leq i\leq s} c_i)$.

Overall complexity

In the rest of this paper we denote by `static1` the static heuristic presented above. Its complexity is

$$O\left(n|\mathcal{E}| + s^3 + n\log n + n(\Delta T \cdot \Delta\mathcal{P} + \Delta T \log(\Delta T)) + ms^2 + n \max_{1\leq i\leq s} c_i\right)$$

which is an order of magnitude less than the complexity of the `min-min` scheme: we no longer have a n^2 term. Here the *closest* policy only induces a complexity increase of $O(ms^2)$ as, in the worst case, each file is needed on all servers, and to find the closest server, one must search them all.

5.2 Variants of the static heuristics

Critical path scheduling of communications: variant `critic`

Instead of scheduling the communications task by task, we schedule at once all the necessary communications. This way, we hope to be able to give a higher priority to the most important communications.

Consider the transfer of a file F from a server S_i to a server S_k, through a single other server S_j. The transfer is thus composed of two local communications, the second ($S_j \xrightarrow{F} S_k$) depending upon the first ($S_i \xrightarrow{F} S_j$). To schedule the whole set of communications, we build the dependence graph of the local communications where each vertex represents a local communication, and where there is an edge from a first vertex to a second if and only if the first one represents a local communication that must have been realized before the second can occur. In our example, there would be an edge from the vertex representing $S_i \xrightarrow{F} S_j$ to the vertex representing $S_j \xrightarrow{F} S_k$. Our dependence graph is obviously a directed acyclic graph (DAG). It contains at most $O(ms)$ vertices (at the end, at worst each file is on each server).

We associate to each vertex $S_i \xrightarrow{F} S_j$ of the dependence graph a *weight* equal to the computation time of the tasks mapped to the cluster C_j and depending on the file F, plus the time of the communication itself. We define the *cost* of a vertex as the sum of its weight and of the weights of all the vertices reachable from it. Finally, in a critical path approach, we schedule the communications in decreasing order of their associated vertices *costs*.

The complexity of this whole communication scheduling is $O(ms\log(ms))$ (note that the costs are computed in $O(ms)$ through a simple graph traversal). Of course, to reach such a low complexity, we assume, as we did for the `min-min` heuristic, that new communications are always scheduled after previously scheduled communications which use the same physical links.

Highest priority to *ready* tasks: variant `readiness`

In the basic version of the heuristic, tasks are selected by following strictly the increasing order on the task costs (Step 5 of Algorithm 3). In the `readiness` variant, each time we try to schedule a new task, we consider the next task in this order, *unless* there is a task T_i which is *ready* for a server S_j, in which case we immediately schedule T_i on S_j. A task is called *ready* for a server, if the server repository holds all the files that the task depends upon. So, a ready task can be scheduled at no communication cost.

Maintaining lists of ready tasks costs $O(s|E|)$. Indeed, on each server, we maintain for each task the number of its missing files. Each time a new file arrives on a server, we decrease the number of missing files for all tasks that depend upon it.

Postponing the mapping of tasks to servers: the `static2` heuristic

In the `static1` heuristic, a task is mapped on a server on which its *cost* is minimal. Following ideas in [9], we only use the *cost* to determine in which order the tasks are considered. Thus, we sort, for each server, the tasks by increasing cost. Once the sorted lists are computed, we still have to map the tasks to the servers and to schedule them. The tasks are scheduled one-at-a-time. When we want to schedule a new task, on each server S_i we evaluate the completion time of the first task (according to the sorted list) which has not yet been scheduled. The completion time evaluation is identical to the equivalent evaluation performed by the `min-min` heuristic. Then we pick the pair task/server with the lowest completion time. This way, we obtain our second static heuristic (see Algorithm 4), which includes by default the *readiness* variant. We denote this heuristic `static2` in the rest of this paper. As `static1`, `static2` follows the *closest* policy of Section 4.6.

The overall complexity of the `static2` heuristic is:

$$O\left(n|\mathcal{E}| + s^3 + sn\log n + s|\mathcal{E}| + ns\Delta T(\Delta\mathcal{P} + s + \log(\Delta T)) + n\max_{1\leq i\leq s} c_i\right)$$

5.3 Dynamic heuristics

In our static heuristics, we first define the order in which the tasks are considered, and all the other scheduling decisions (communication definition and scheduling) are implied by this original order. However, this order is based on a cost which is truly relevant when there is a single task in the system. Indeed, the cost formula does not take into account the fact that some other tasks may need the same files, and thus misses the possibility of new sources (created as the execution proceeds) for these files. To remedy this flaw, we introduce a more dynamic scheme, while trying to conserve a low complexity heuristic. The structure of our dynamic heuristics is described by Algorithm 5.

A dynamic cost

Our dynamic cost function is built along the line of the cost defined by Equation 1 using the simple communication estimation of Equation 2 and replacing the constant "source" function by a dynamic "closest" function:

$$a(T_i, S_j) = Op_{e_{k,i}\in\mathcal{E}}\ comm(closest(F_k, S_j)) + \frac{t_i}{\sum_{1\leq l\leq p(j)} s_{j,l}} \tag{3}$$

Algorithm 4 Structure of the `static2` heuristic

1: **for** each server S_j **do**
2:   **for** each task T_i **do**
3:     compute the cost $a(T_i, S_j)$
4:   build the list $L(S_j)$ of the tasks sorted in increasing value of $a(T_i, S_j)$
5: **while** there remains tasks to schedule **do**
6:   **for** each server S_j **do**
7:     **if** there are tasks ready for S_j **then**
8:       let T_i be any task ready for S_j
9:     **else**
10:       let T_i be the first unscheduled task in $L(S_j)$
11:     use Algorithm 1 to compute the date t at which all the files required for the execution of T_i will be available on S_j
12:     evaluate the minimum completion time of T_i on S_j knowing t (as computed above) and considering the cluster C_j as a single processor
13:   pick a couple (S_j, T_i) whose minimum completion time is minimal
14:   map T_i on the cluster C_j
15:   use Algorithm 1 to schedule the communications needed by the execution of T_i on S_j
16: **for** each server S_j **do**
17:   greedily schedule the tasks mapped on S_j

Algorithm 5 Structure of the dynamic heuristics

1: **for** each task T_i **do**
2:   **for** each server S_j **do**
3:     compute the cost $a(T_i, S_j)$
4: **while** there are tasks to be scheduled **do**
5:   **for** each server S_i **do**
6:     pick the task(s) of lowest dynamic cost(s) on S_i
7:   among the tasks picked at Step 5, select the couple(s) (S_i,T_j) of lowest dynamic cost(s), taking into account the amount of work already assigned to cluster C_i
8:   **for** each selected couple (S_i,T_j) in any order **do**
9:     map T_i on the cluster C_j
10:     use Algorithm 1 to schedule the communications needed by the execution of T_i on S_j
11:     Re-evaluate the costs whose value may have changed
12: **for** each server S_j **do**
13:   greedily schedule the tasks mapped on S_j

The "closest" function, when invoked on a file F and a server S, returns the identity of the server which is closest to S and which holds the file F *at the time the function* was invoked. In other words, the "closest" function takes into account the mapping decisions taken before its invocation.

A new problem arises: the ability to compute the "closest" function at the lowest possible cost. To reach this goal, each server holds a table of the other servers sorted in increasing distance (in communication time). Building such a list costs the computation of all the pairwise distances plus the sorting of the s tables, that is $O(s^3 + s^2 \log s) = O(s^3)$. Each time a new communication is decided involving a file F, the value of $closest(F, S_i)$ is recomputed for any server S_i. This is done in constant time by checking whether the new server holding F has a lower rank, in the table defined above, than the previous *closest* server for this file. As, in the worst case, each file is only sent once to each of the servers, there are at most $O(sm)$ of these constant-time updates.

Each time the value of a function $closest(F_k, S_j)$ changes, we recompute the value of $a(T_i, S_j)$ if and only if task T_i depends upon file F_k, i.e. $e_{k,i} \in \mathcal{E}$. Any of these updates is in constant time if the operator "Op" is the sum operator, and cost $O(\Delta T)$ if "Op" stands for "max". Therefore, the overall complexity of maintaining our dynamic costs is

$$O(s^3 + sm(1 + C))$$

where $C = 1$ if "Op" is the sum operator and $C = \Delta T$ if "Op" is the max operator.

Picking the next task(s) to be scheduled

Once we have defined our dynamic cost, we have to decide how to use it to select the new task(s) to be scheduled. We have the choice either to pick a single new task at a time or to pick a set of k tasks. The former scheme is in spirit closer to the `min-min` heuristic. The latter scheme may be less expensive. In both versions, we once again target a low complexity.

Heuristic `dynamic1`

A simple way of picking a single new task would be to search, on each server, which task has the lowest complexity, and then search for the minimum over the servers. This is exactly what the `min-min` heuristic does. Such a selection scheme costs $O(n^2 s)$ which may be prohibitive. In order to speed-up the search of the task of lowest cost, we maintain on each server a heap of the task costs. Then, on each server, the selection of the task of lowest cost is done in constant time. However, each time we update a cost, we have to pay an additional cost of $O(\log n)$ for the removal of an element from the heap and the addition of a new one[1]. Therefore, the overall complexity of this heuristic is:

$$O\left(n|\mathcal{E}| + s^3 + sm(1 + C) + s(m + n)\log n + n\Delta T(\Delta\mathcal{P} + s + \log(\Delta T)) + n \max_{1 \le i \le s} c_i\right).$$

Heuristic `dynamic2`

Another way of decreasing the complexity of the selection of the "closest" task candidate is to select, on each server, k tasks of lowest costs, instead of only 1 task. Such a selection can be realized in linear time in the worst case [2, 6]. Therefore, we select the k tasks of lowest cost on each server. As a task may appear in the "closest" set of different servers, we sort the $k.s$ couples (task, server) that we obtain accroding to their costs, and we pick the k distinct tasks of lowest costs. Therefore, the overall complexity of this heuristic is:

$$O\left(n|\mathcal{E}| + s^3 + sm(1 + C) + \frac{n}{k}(ns + ks\log(ks)) + n\Delta T(\Delta\mathcal{P} + s + \log(\Delta T)) + n \max_{1 \le i \le s} c_i\right).$$

[1] The removal of any element in a heap cost $O(\log n)$ if we maintain a table associating any element stored in the heap to its position in the heap. Maintaining such a table does not increase the theoretical complexity of the heap operations.

6 Simulation results

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

6.1 Simulation platforms

Platform graphs: they are composed of 7 servers. The graph of the servers is either a clique, a random tree, or a ring.

Clusters: we have recorded the computational power of 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 realistic and heterogeneous clusters randomly containing 8, 16, or 32 processors.

Communication links: the communication links between the servers are randomly 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 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).

6.2 Task 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 75 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.

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 1500 tasks and 1750 files, except for the fork graphs which also contain 1500 tasks but only 70 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.

The initial distribution of files to server is built randomly.

6.3 Results

Table 1 summarizes all the experiments. In this table, we report the performance of the best ten heuristics, and of the naive *random*, together with their cost (i.e. their CPU time). This is a summary of $36,000$ random tests ($1,000$ tests over all four task graph types, three platform graph types, and three communication-to-computation cost ratios). Each test involves 27 heuristics:

- `min-min` and its variant `min-min+closest`.

Heuristic	Relative performance	Standard deviation	Relative cost	Standard deviation
min-min+closest	1.01	0.04 (3.7%)	4200	790 (19%)
min-min	1.17	0.16 (14%)	3600	820 (23%)
static2	1.28	0.23 (18%)	5.2	1.5 (28%)
static2+max	1.29	0.23 (18%)	5.6	1.3 (23%)
static1+readiness	1.84	0.54 (29%)	1.9	0.6 (30%)
static2+max+critic	1.91	0.64 (33%)	6.4	1.1 (17%)
static2+critic	1.92	0.66 (34%)	5.9	1.2 (20%)
static1	1.93	0.60 (31%)	1.7	0.5 (31%)
static1+readiness+max	2.06	0.83 (40%)	2.0	0.5 (28%)
static1+max	2.08	0.85 (41%)	1.8	0.5 (30%)
random	118	380 (320%)	1.5	0.3 (19%)

Table 1: Relative performance and cost of the best ten heuristics and of the naive *random*.

- The three static heuristics: `static1`, `static1+readiness`, and `static2`. These heuristics are declined with the variants `max` ("Op" is the max operator instead of the sum, cf. Section 5.1) and `critic`.
- The dynamic heuristics `dynamic1` and `dynamic2`, the latter using sets of either $k = 10$ or $k = 100$ tasks. These heuristics are also declined with the variants `max` and `critic`.
- The naive `random` heuristic, which randomly picks the next task to be scheduled and the server on which to execute it. Once these decisions are randomly taken, the communication and tasks are scheduled as in the `static1` heuristic. We include this naive heuristic as a lower bound on performance.

For each of the $36,000$ tests, we compute the ratio of the performance of all heuristics over the best heuristic for the test, 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 $36,000$ tests, the best heuristic for this particular case. (For each test, the relative cost is computed along the same guidelines, using the fastest heuristic.)

Figures 8 through 13 present detailed comparisons of all the heuristics, by types of task graphs, by communication to computation ratio, or by types of platform graphs.

From all these results, we can see that the two versions of `min-min` are the best heuristics, and that `min-min+closest` is almost always the best heuristic. We see that the *closest* variant as a great beneficial impact of about 15% in performance for a reasonable overhead cost of 16%. This is the reason why we use by default the *closest* variant in the `static1` and `static2` heuristics.

As observed in [9], the *readiness* variant has a significant impact on performance. For the `static1` heuristic, it induces a performance gain of about 5% for an overhead cost of 10%. So, we use it by default in the `static2` heuristic.

The four versions of the `static2` heuristics are in the top ten. Furthermore, the two basic versions are within 28% of the best heuristic for... 0.1% of its cost! Surprisingly, the way in which sets of communications are approximated in the cost function has almost no influence on the quality of the output. Indeed, there is no significative performance difference between the version `static2`, which over-pessimistically approximates the sets of communications, and the version `static2+max`, which over-optimistically approximates them. This certainly emphasizes the impact of the precise evaluation of the minimum completion time which is used to decide which task is scheduled next. This precise evaluation induces a 30% improvement in performance (when comparing to `static1+readiness`) but at the price of an overhead cost of about 220%.

None of our dynamic heuristics made the top ten. The first one is ranked 14: this is the dynamic1+max heuristic. Its relative performance equals 2.43, which is almost twice as worse as static2, and its relative cost is 14. The dynamic2 heuristic using sets of 10 tasks ranks 15 with a relative performance of 2.51 and a relative cost of 26 (increasing the size of sets to 100 tasks, the relative performance slightly increases to 2.54 while the relative cost drops to 5.2). These results are rather disappointing. We were able to design dynamic heuristics of reasonable complexities, but there performances are quite low. As our decision process does not take into account the amount of communications already scheduled, and their destinations, our dynamic heuristics may tend to map all the tasks on the same processor. This would obviously lead to network congestion.

Looking at Figures 10 and 11, we see that our heuristics have the same behavior whatever the communication to computation ratio is. Furthermore, our best heuristic, static2, has almost exactly the same relative performance whatever this ratio is. This shows the robustness of these heuristics.

Finally, we introduced the naive random heuristic as a lower bound on the quality of the task mapping and scheduling. Figures 8 and 9 show that the random heuristic is extremely bad on the *fork* graphs. It appears that when the task graph contains evident sharing properties, as in the *fork* graphs, our heuristics are able to take advantage of them. On the contrary, on the *fork* graphs, random is as bad as usual and the gap widens. On the other types of graphs, the performance of random is not as bad as one could have expected. This may be due to our choice to have a random initial distribution of files to servers. Our simulation systems may be too general, and thus too difficult, for any heuristics to have tremendous performance. Also, our simulation platforms may be small enough to limit the impact of the randomness of random. Nevertheless, our best heuristics still show significant improvements in comparison to random, while this is not always the case for the dynamic heuristics (not a good mark for them!). We ran the static2 and random heuristics on a larger system of 20 servers, 50000 tasks, and 62500 files. Table 2 shows the relative performance (the lower the numbers in the table, the better the static2 heuristic) of static2 when compared to random on the original and on the larger system. We clearly see that in comparison the relative performance of static2 improves when the system become larger.

Task graph	Original system	Larger system
Two-one	0.36	0.17
Random	0.31	0.29
Partitioned	0.57	0.37
Forks	0.002	0.00005

Table 2: Relative performance of static2 in comparison to random.

7 Conclusion

In this paper, we have dealt with the problem of scheduling a large collection of independent tasks, that may share input files, onto collections of distributed servers. On the theoretical side, we have shown a new complexity result, that shows the intrinsic difficulty of the combinatorial problem of deciding where to move files.

On the practical side, our contribution is twofold:

- We have shown how to extend the well-known min-min heuristic to the new framework; this turned out to be more difficult than expected, and we had to introduce (and justify) several restrictive assumptions upon the routing and communication scheduling.
- We have succeeded in designing a collection of new heuristics which have reasonably good performance but whose computational costs are an order of magnitude lower than min-min. Specifically, the best heuristic static2 produces schedules whose makespan is 30% higher than min-min, but whose cost (in CPU time) is 800 times smaller.

We plan to deploy the heuristics presented in this paper for a large medical application, with servers in different hospitals in the Lyon-Grenoble area, and we hope that the ideas introduced when designing our heuristics will prove useful in this real-life scheduling problem.

References

[1] 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.

[2] M. Blum, R. W. Floyd, V. Pratt, R. R. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Technical 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] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[8] 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.

[9] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *PDP'2004, 12th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE Computer Society Press, 2004.

[10] 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.

[11] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.

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

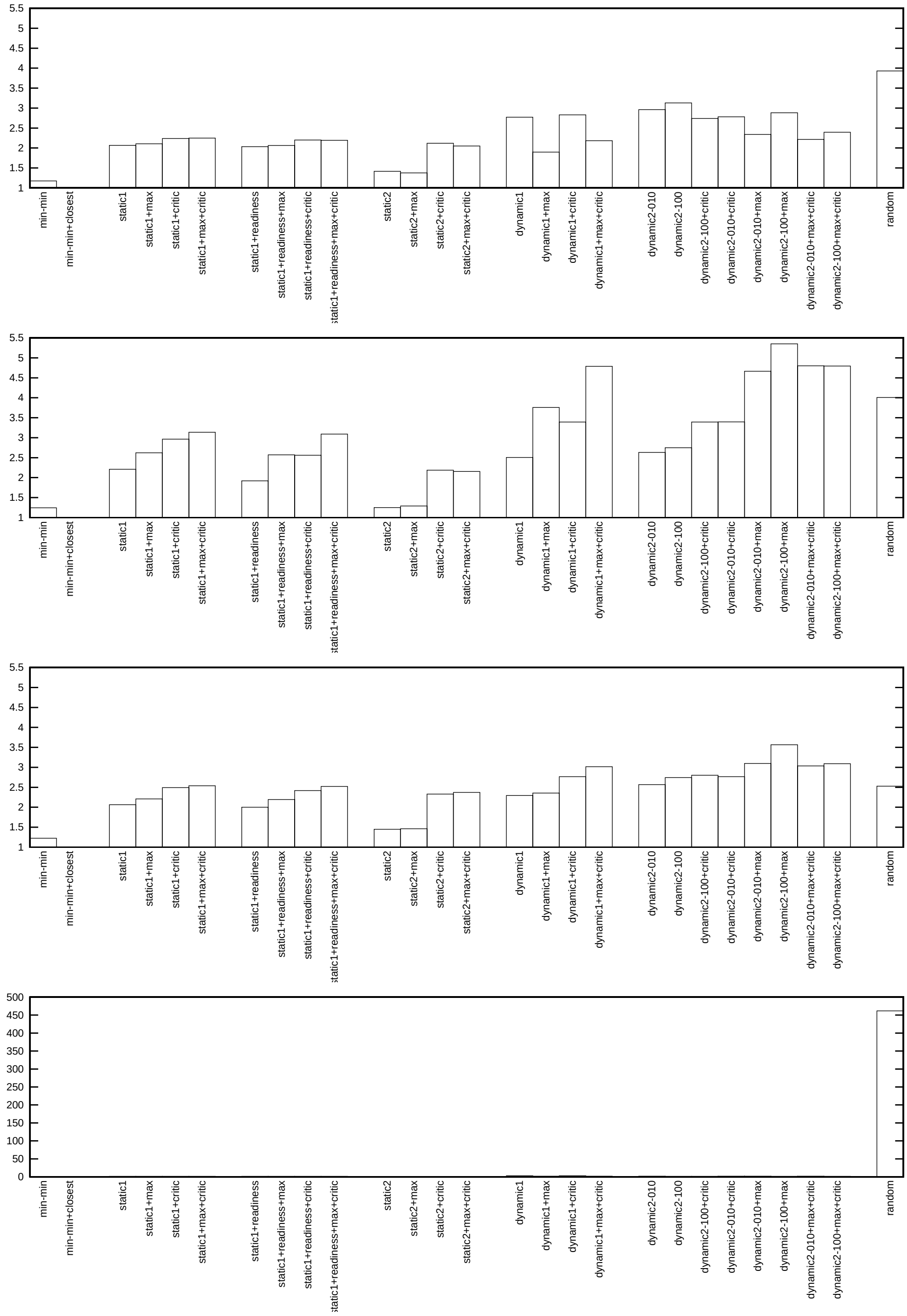

Figure 8: Relative performances of the schedules produced by the different heuristics average on three communication to computation ratios and three types of platform graphs, for the four types of graphs (from top to bottom: *Two-one*, *Random*, *Partitioned*, and *Forks*).

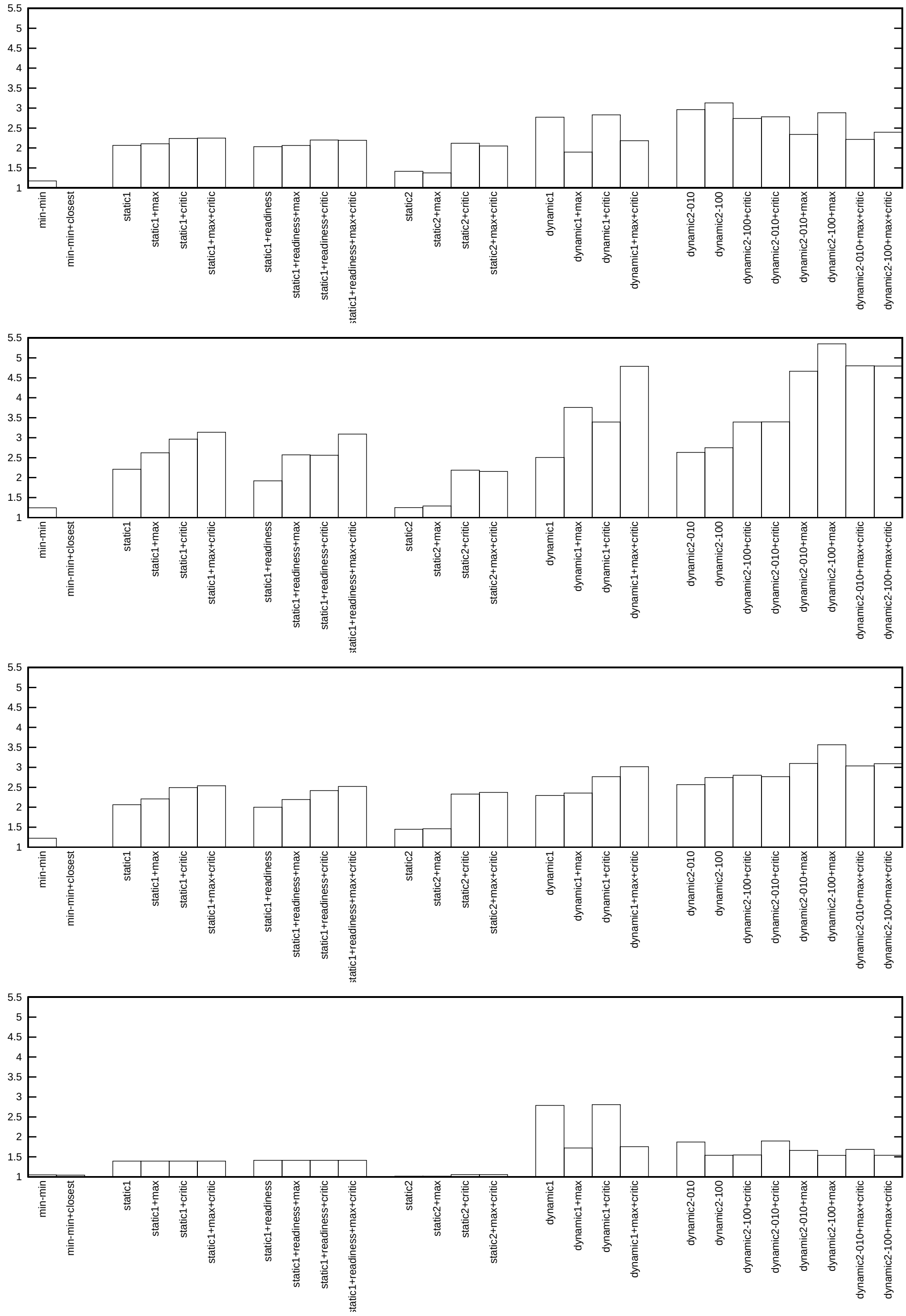

Figure 9: Relative performances of the schedules produced by the different heuristics *(except random)* average on three communication to computation ratios and three types of platform graphs, for the four types of graphs (from top to bottom: *Two-one*, *Random*, *Partitioned*, and *Forks*).

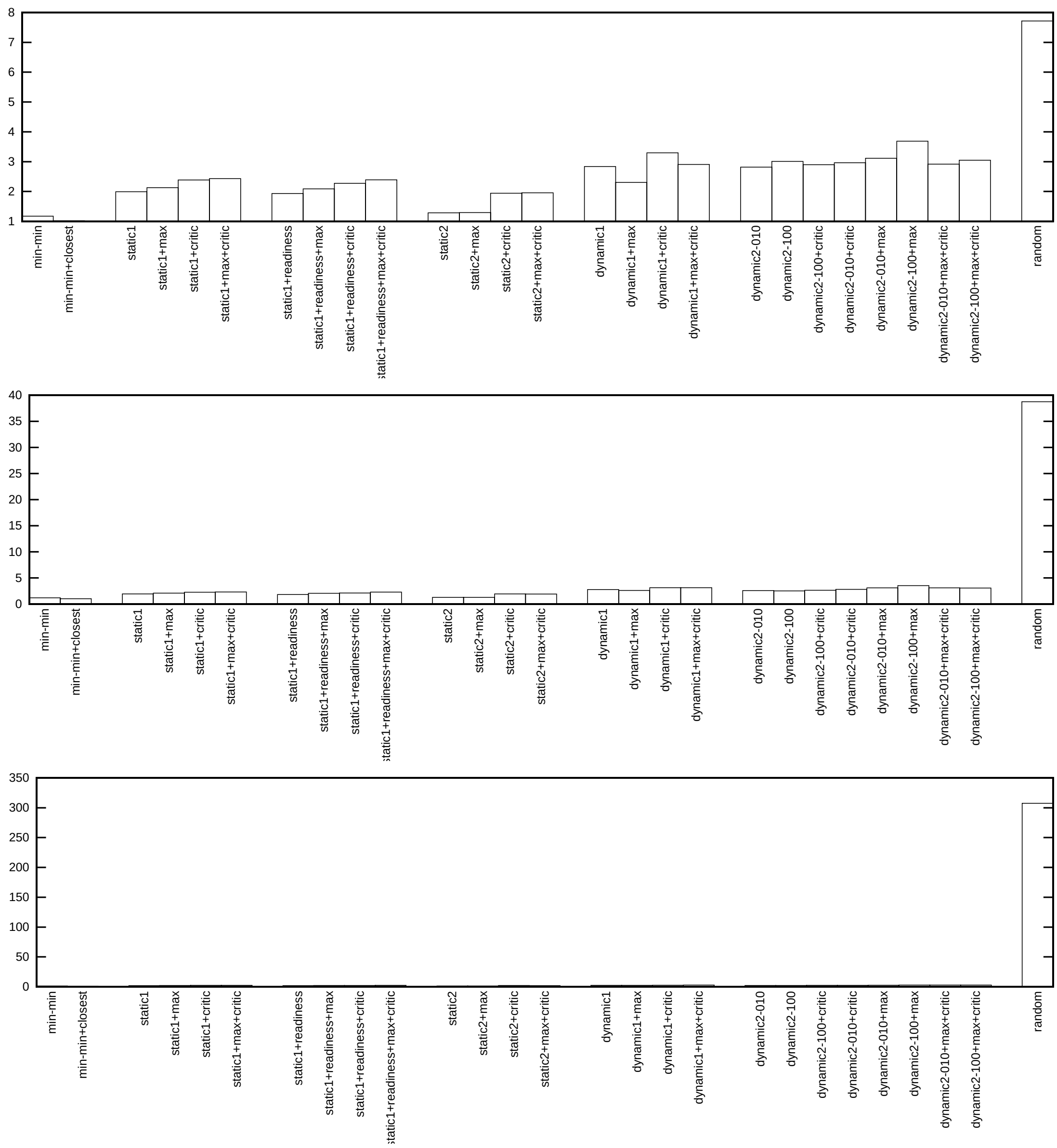

Figure 10: Relative performances of the schedules produced by the different heuristics average on the four types of graphs and three types of platform graphs, with a communication to computation ratio equal, from top to bottom, to: 0.1, 1.0, and 10.

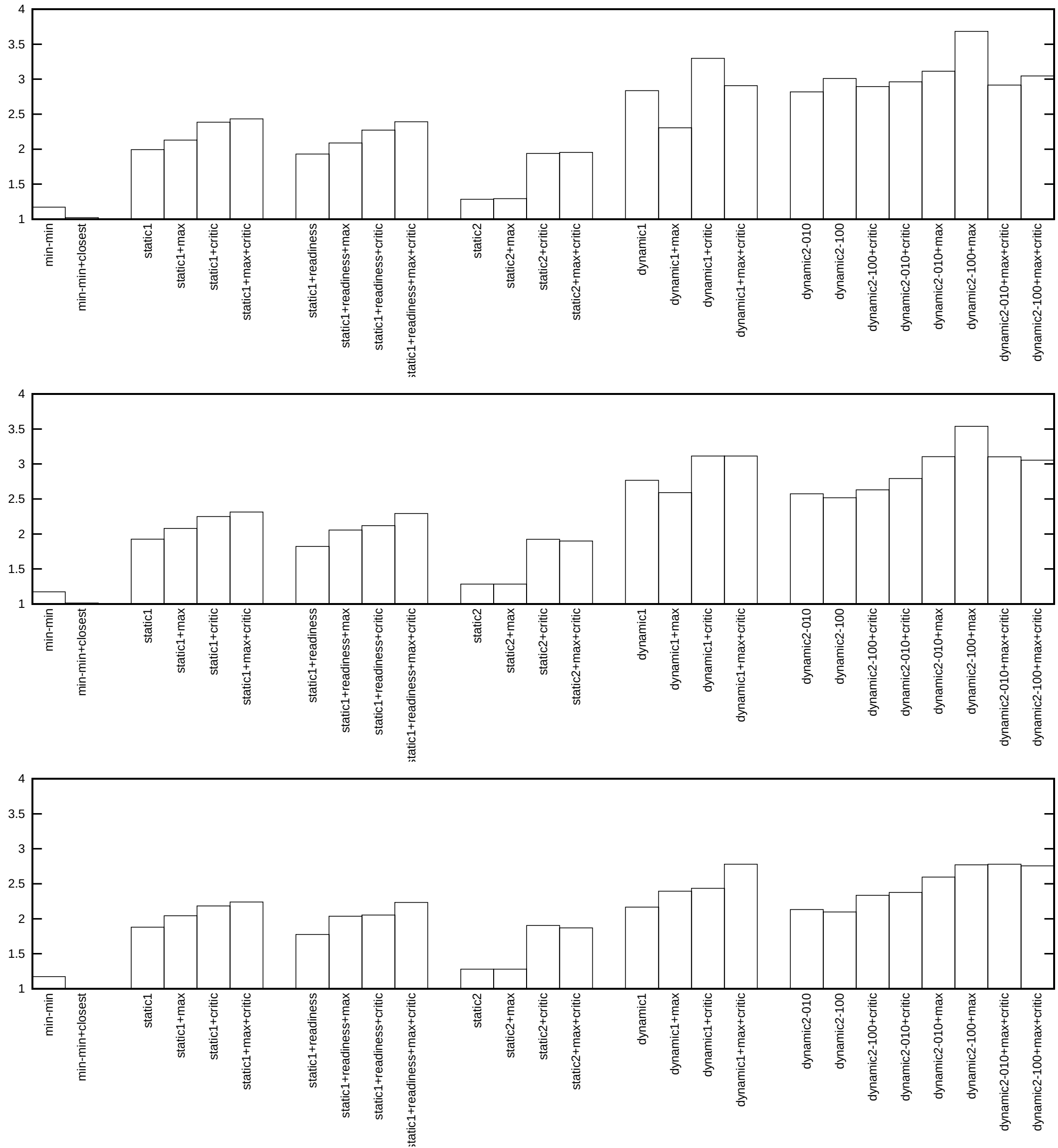

Figure 11: Relative performances of the schedules produced by the different heuristics *(except random)* average on the four types of graphs and three types of platform graphs, with a communication to computation ratio equal, from top to bottom, to: 0.1, 1.0, and 10.

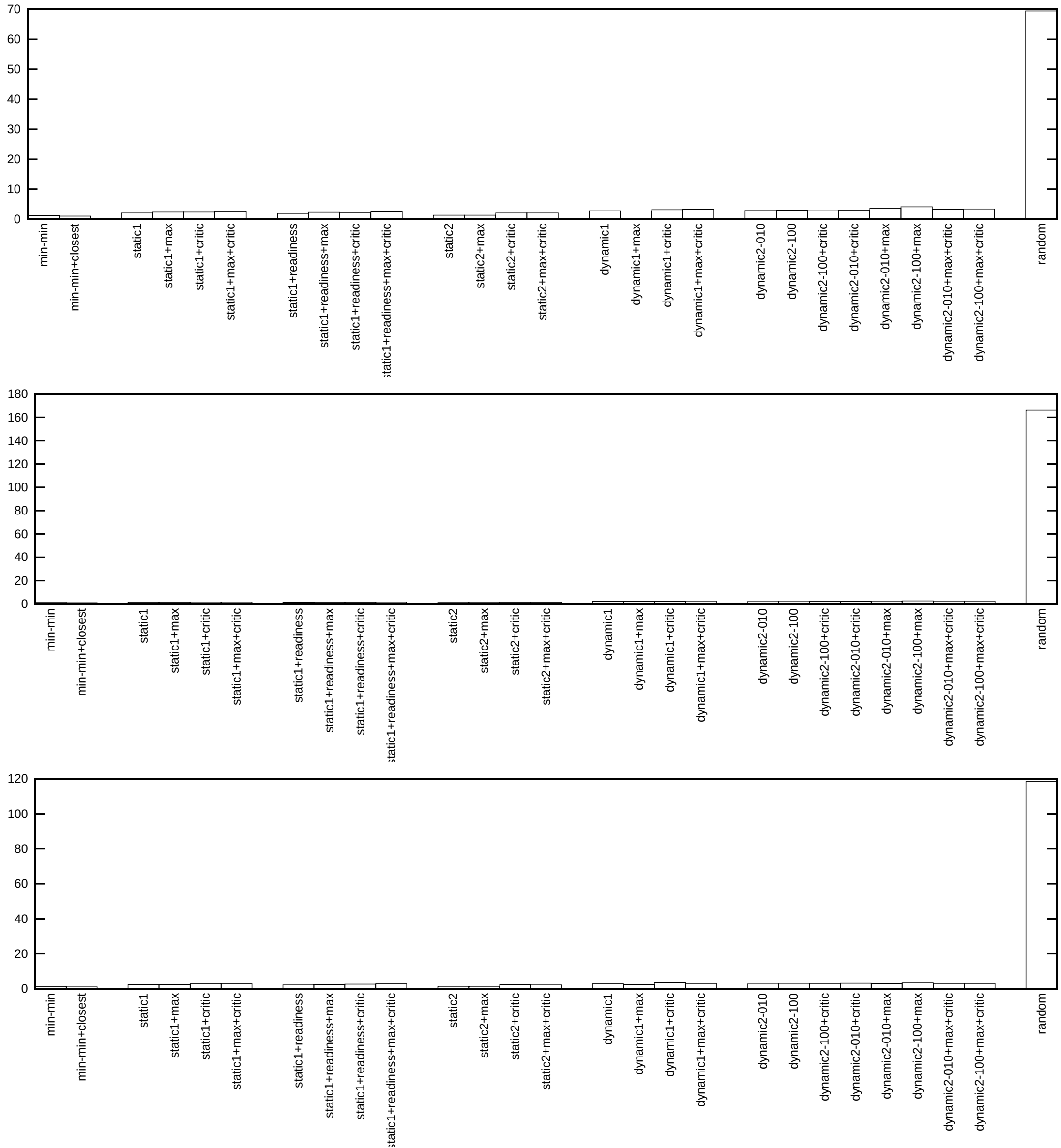

Figure 12: Relative performances of the schedules produced by the different heuristics average on the four types of graphs and three communication to computation ratios, for the three types of platform graphs, from top to bottom: clique, random tree, and ring.

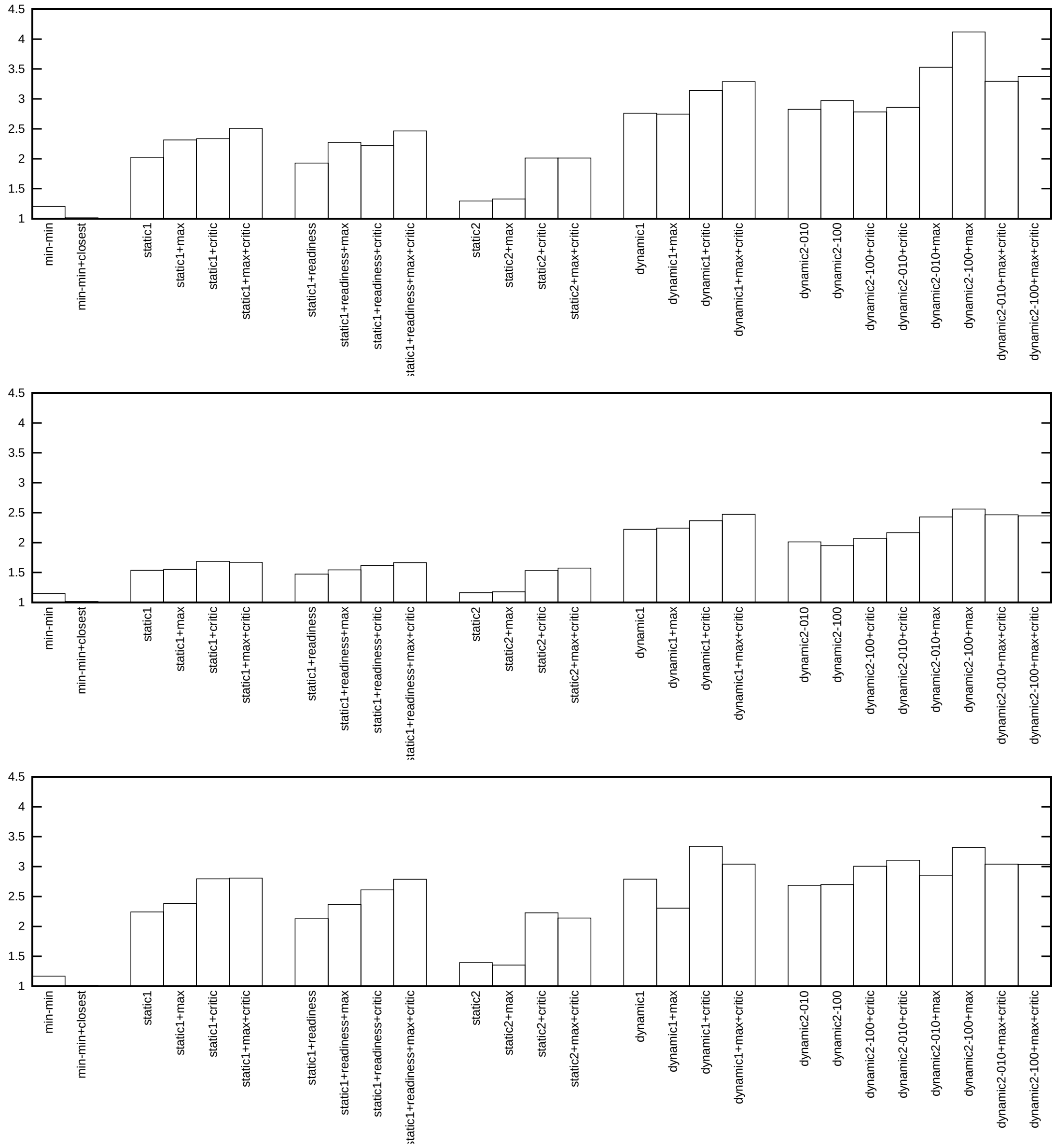

Figure 13: Relative performances of the schedules produced by the different heuristics *(except random)* average on the four types of graphs and three communication to computation ratios, for the three types of platform graphs, from top to bottom: clique, random tree, and ring.