Scheduling Tasks Sharing Files from Distributed Repositories

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, Lyon, France

Abstract. This paper is devoted to scheduling a large collection of independent tasks onto a 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 which server will execute it, and to transfer the required files 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 a complexity result that assesses the difficulty of the problem. On the practical side, we design several new heuristics, including an extension of the `min-min` heuristic to such a decentralized framework, and several lower cost heuristics, which we compare through extensive simulations.

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 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] on one hand, and by Giersch, Robert, and Vivien [5] on the other hand. In [3], Casanova et al. target the scheduling of the tasks typically submitted to APST, the AppLeS Parameter Sweep Template [1]. Casanova et al. have considered three heuristics designed for completely independent tasks (no input file sharing) that were proposed in [6]. They have modified these three heuristics (originally called `min-min`, `max-min`, and `sufferage` in [6]) to adapt them to the additional constraint that input files are shared between tasks. 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.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 246–253, 2004.

In [5], Giersch et al. have introduced several new heuristics, which are shown to perform as efficiently as the best heuristics in [3] 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 master distributes the files to the processors, so that they can execute the tasks. 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 the local repository of S_i 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 in Section 3 a complexity result that assesses the difficulty of our scheduling problem. On the practical side, we design several heuristics. The first heuristic is the extension of the `min-min` heuristic to the decentralized framework (Section 4). The next heuristics aim at retaining the good performances of the `min-min` variants while reducing the computational cost by an order of magnitude (Section 5). We compare all these heuristics through extensive simulations (Section 6). We start by describing in Section 2 the specifications of our scheduling problem.

Due to space limitations, we refer to [4] for missing details and proofs.

2 Framework

Tasks and files. The problem is to schedule a set of n independent tasks $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. The weight of task T_j is t_j, $1 \leq j \leq n$. 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. Each node in $\mathcal{V} = \mathcal{F} \cup \mathcal{T}$ is weighted by f_i or t_j.

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 $\mathcal{S} = \{S_1, \ldots, S_s\}$ 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 linking any server pair.

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 are stored in the repositories. We assume that a file may be duplicated, and thus simultaneously stored on several 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.

For cluster C_i to be able to process task T_j, repository R_i must hold all files that T_j depends upon. 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 F_k such that $e_{k,j} \in \mathcal{E}$, 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, first consider the case of adjacent servers in the platform graph. Suppose that server S_i sends the file F_j to another server S_k, through 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 communication cost is the sum of the costs of the adjacent communications. Leaving copies of transferred files on intermediate servers multiplies the 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 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.

As for computation costs, each cluster C_i is composed of c_i heterogeneous processors $C_{i,k}$, $1 \le k \le c_i$. 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 and less precise approach is to view cluster C_i as a single computational resource of cumulative speed $\sum_{k=1}^{c_i} s_{i,k}$. This is the approach used in all of our heuristics when the completion time of a task on a server needs to be evaluated.

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 two important points: (i) some files may well be sent several times, so that several clusters can independently process tasks that depend upon these files, and (ii) a file sent to some repository remains available for the rest of the schedule (if two tasks depending on the same file are scheduled on the same cluster, the file must only be sent once).

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

3 Complexity

Most scheduling problems are known to be difficult [7]. 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 (it reduces to the 2-PARTITION problem as 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 [5]. Even in the un-weighted version of our problem, where all files have same size and all communication links have same bandwidth, to decide where to move the files so as to execute the tasks is a difficult combinatorial problem due to file sharing. This is what formally state Definition 1 and Theorem 1.

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: (i) uniform file sizes ($f_i = 1$), (ii) homogeneous interconnection network ($b_i = 1$), and (iii) 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.*

4 Adapting the `min-min` Scheme

Due to the presence of weights in tasks, in files, and in the platform graph, approximation algorithms are not likely to be feasible. Hence, we look for polynomial heuristics to solve TSFDR, and we will compare these heuristics through extensive simulations. Considering the work of Casanova et al. [3] for master-slave systems with a single server, we start by adapting 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 task T_k that remains to be scheduled and each processor $C_{i,j}$, evaluate the Minimum Completion Time (MCT) of T_k if mapped on $C_{i,j}$;
2. pick a couple $(C_{i,j}, T_k)$ with minimum MCT and schedule T_k on $C_{i,j}$.

The problem then is to evaluate a task 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 brought from other servers. One can easily determine which communications should take place. But scheduling these communications is an NP-complete problem in the general case.

Scheduling the communications. We deal with the situation where all communications have the same destination (namely the server that will execute the candidate task T_k). In the 1-port model, if the routing in the platform graph is not fixed, we show that scheduling a set of communications is already NP-hard [4] in our context. As the routing is usually decided by table lookup, one can assume the routing to be fixed. But when trying to schedule the communications required for a task T_k, one must take into account that the communication links are already used at certain time slots due to previously scheduled communications. Even under a fixed routing, this also leads to an NP-complete problem [4].

Therefore, we rely on heuristics to schedule the necessary communications. A first idea would be to memorize for each link the date and length of all communications already scheduled, and to use an *insertion scheduling* scheme to schedule new communications. This scheme should be rather precise but may be very expensive. A second idea is to use a simple *greedy* algorithm which schedules new communications as soon as possible but always *after* communications using the same links. In both cases we assume a fixed routing. If a file is available on several servers, we heuristically decide to bring it from the closest server.

Complexity of the adapted `min-min` scheme. The complexity of the whole `min-min` heuristic is $O(n^2 s \Delta T(s + \Delta\mathcal{P} + log(\Delta T)) + n \max_{1 \leq i \leq s} c_i)$, when using the greedy communication scheduling heuristic, if we denote by ΔT the maximum number of files that a task depends upon, and by $\Delta\mathcal{P}$ the diameter of the platform graph. Of course, the complexity is larger with the insertion scheduling scheme. The last term in the expression comes from scheduling tasks on clusters. Indeed, once the communications are scheduled, 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 reach its MCT.

The `sufferage` heuristic is a variant of `min-min` which sometimes delivers better schedules, but whose complexity is slightly greater [3,5,4].

5 Heuristics of Lower Complexity

As appealing as the `min-min` scheme could be because of the quality of the scheduling it produces [5], 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.

The principle of our heuristics is quite simple. The `min-min` scheme is especially expensive as, each time it attempts to schedule a new task, it considers all the remaining tasks and compute their MCTs. On the opposite, we worked on solutions where we only consider a single task candidate per cluster. This leads to the following scheme:

While there remain tasks to be scheduled do
1. for each cluster C_i pick the "best" candidate task T_k that remains to be scheduled;
2. pick the "best" couple (C_i, T_k) and schedule T_k on C_i.

The whole heuristic then relies on the definition of the "best" candidate. For that, we design a *cost* function that we use as an estimate of the MCT of a task on a given server. Using the results of [5], we simply defined the *cost* of a task T_i on a server S_j as the sum of the time needed to send the required files to S_j (communication time), plus the time needed by the cluster C_j to process the task, when the cluster is seen as a single computational resource (computation time). The communication time can be evaluated using either of the two heuristics described in Section 4. In practice, we approximate it by using either the "sum" of all the communications required (over-approximation by sequentialization of all communications) or its "max" (under-approximation by considering all communications to take place in parallel).

Static heuristics. In our static heuristics, for each cluster we first build a list of all tasks sorted by increasing *cost* function. Then, each time we schedule a new task: 1) we define as local candidate for cluster C_i the task which has lowest *cost* on C_i and has not already been scheduled; 2) among all the local candidates we take the one of lowest *cost* and we assign it on the corresponding cluster; 3) we schedule the necessary communications and the computation as we did for the `min-min` scheme. The overall complexity of this scheme is: $O(n|\mathcal{E}| + s^3 + sn\log n + n\Delta T(\Delta\mathcal{P} + \log(\Delta T)) + ms^2 + n\max_{1\leq i\leq s} c_i)$ (with greedy communication scheduling) which is an order of magnitude less than the complexity of the `min-min` scheme: we no longer have a n^2 term.

In the MCT variant of our static heuristic we make two modifications to our scheme. First, if on one cluster there is a task which only depends on files residing on the corresponding server, it becomes the local candidate (whatever its *cost*). Then, we compute the actual MCT of each local candidate (only on its candidate cluster) and we pick the task of lowest MCT. The two modifications only lead to an additional term of $O(ns\Delta T)$ in the complexity.

Dynamic heuristics may seem a better approach than static ones. Indeed, as scheduling decisions are taken, files are copied between servers and the original *cost* estimate become less pertinent. Hence the desire to use dynamically updated *costs* to drive the mapping decisions. But the updates should be kept as computationally cheap as possible. Hence, each time a file F_j is duplicated we update only the *costs*(C_i, T_k) where T_k depends on F_j. Using a clever data structure (see [4] for details) the overall complexity of maintaining dynamic *costs* is only $O(s^3 + sm(1+u))$ where $u = 1$ when we over-approximate the communications and $u = \Delta T$ when we under-approximate them.

In the `dynamic1` heuristics, for each cluster we maintain a heap of dynamic *costs*, and the selection of the local candidate for each cluster is cheap. Then, the overcost due to the dynamicity is only: $O(sm(1+u) + sm\log n + ns\Delta T)$.

Another way of decreasing the complexity of the selection of the local candidate is to select, on each server, k tasks of lowest *costs*, instead of only one task. Such a selection can be realized in linear time in the worst case [2]. The `dynamic2` heuristic uses this approach. Then, the overcost due to the dynamicity is: $O(sm(1+u) + \frac{n}{k}(ns + ks\log(ks)) + ns\Delta T)$. For the simulations described in the next section, we used $k = 10$.

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.

Simulation framework. We generated three types of server graphs (7 servers): clique, random tree, or ring. Each server is associated a cluster of 8, 16, or 32 processors. The processors speeds and communication links bandwidths were chosen from a set of values recorded on real machines. These values were normalized so as to model three communication-to-computation cost ratios: computation intensive, communication intensive, and intermediate. For the tasks graphs, we generated four types of graphs, from very regular ones with lots of file sharing, to fully randomly built ones. The initial distribution of files to server is built randomly. A fully detailed description of the simulations can be found in [4].

Results. We report the performance of our heuristics together with their cost (i.e., their CPU time). The `randommap` heuristic randomly picks the local candidate (the same for all clusters) but uses the same optimizations and scheduling schemes than the other heuristics. Table 1 summarizes all the simulations, i.e., 36,000 random tests (1,000 tests over each combination of task graph, platform graph, and communication-to-computation cost ratio). For each test, 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 of 1. 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. (The *relative cost* is computed along the same guidelines, using the fastest heuristic.)

The basic versions of our heuristics are far quicker than the `min-min` versions but at the cost of a great loss in the quality of the schedules produced (two times worse). The MCT variant greatly improves the quality of our heuristics (this is exemplified by `randommap`) while their costs remain very low. For example, the

Table 1. Relative performance and cost of the heuristics: basic versions, MCT variants, and MCT variants with communication scheduling with insertion scheduling. Standard deviations are in parentheses (for relative costs, all are between 128% and 211%).

Heuristic	Basic version		MCT variant		MCT variant + insert.	
	Perf.	Cost	Perf.	Cost	Perf.	Cost
`min-min`	1.14 (± 7.9%)	31,050	-	-	1.08 (± 7.5%)	61,771
`sufferage`	1.16 (± 14%)	33,985	-	-	1.07 (± 12%)	77,991
`static`	2.20 (± 33%)	16	1.46 (± 23%)	44	1.18 (± 11%)	56
`dynamic1`	2.32 (± 37%)	42	1.35 (± 17%)	67	1.11 (± 8.8%)	77
`dynamic2`	2.42 (± 39%)	310	1.92 (± 36%)	82	1.40 (± 31%)	90
`randommap`	141 (± 317%)	11	1.32 (± 18%)	41	1.08 (± 6.9%)	53

MCT variant of `dynamic1` produces schedules which are only 19% longer than those of `min-min`... but it produces them 460 times more quickly. We also ran the heuristics with the insertion scheduling heuristic for communication scheduling (rather than with the greedy scheduling as previously). As predicted, the quality of results significantly increased. The overhead is prohibitive for the `min-min` variants (these versions only differ from the original by the insertion scheduling scheme). Surprisingly, this overhead is reasonable for our heuristics. For example, this version of `dynamic1` produces schedules which are 3% *shorter* than those of the original `min-min`... and it produces them 400 times more quickly.

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: 1) we have shown how to extend the well-known `min-min` heuristic to this new framework, which turned out to be more difficult than expected; 2) we have succeeded in designing a collection of new heuristics which have reasonably good performance but whose computational costs are orders of magnitude lower than `min-min`: our best heuristic produces schedules whose makespan is only 1.1% longer than those of the best `min-min` variant, and produces them 585 times faster than the quickest `min-min` variant.

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*. 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. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *HCW'2000*. IEEE Computer Society Press, 2000.
4. A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files from distributed repositories (revised version). Research Report RR-2004-04, LIP, ENS Lyon, 2004.
5. A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *PDP'2004*. IEEE Computer Society Press, 2004.
6. M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, Nov. 1999.
7. B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.