Scheduling Tasks Sharing Files on Heterogeneous Clusters

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

[1] ICPS/LSIIT, UMR CNRS-ULP 7005, Strasbourg, France
[2] LIP, UMR CNRS-INRIA 5668, École Normale Supérieure de Lyon, France

Abstract. This paper is devoted to scheduling a large collection of independent tasks onto heterogeneous clusters. The tasks depend upon (input) files which initially reside on a master processor. A given file may well be shared by several tasks. The role of the master is to distribute the files to the processors, so that they can execute the tasks. The objective for the master is to select which file to send to which slave, and in which order, so as to minimize the total execution time. The contribution of this paper is twofold. On the theoretical side, we establish complexity results that assess the difficulty of the problem. On the practical side, we design several new heuristics, which are shown to perform as efficiently as the best heuristics in [3, 2] although their cost is an order of magnitude lower.

1 Introduction

In this paper, we are interested in scheduling independent tasks onto heterogeneous clusters. These independent tasks depend upon files (corresponding to input data, for example), and difficulty arises from the fact that some files may well be shared by several tasks. This paper is motivated by the work of Casanova et al. [3, 2], who target the scheduling of tasks in APST, the AppLeS Parameter Sweep Template [1]. Typically, an APST application consists of a *large* number of independent tasks, with possible input data sharing. 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. [3, 2] have considered three heuristics designed for completely independent tasks (no input file sharing) that were proposed in [5]. They have modified these three heuristics (originally called Min-min, Max-min, and Sufferage in [5]) to adapt them to the additional constraint that input files are shared between tasks. As was already pointed out, the number of tasks to schedule is expected to be very large, and special attention should be devoted to keeping the cost of the scheduling heuristics reasonably low.

We restrict to the same special case of the scheduling problem as Casanova et al. [3, 2]: we assume the existence of a master processor, which serves as the repository for all files. The role of the master is to distribute the files to the processors, so that they can execute the tasks. The objective for the master is to select which file to send to which slave, and in which order, so as to minimize the total execution time. The contribution of this paper is twofold. On the theoretical side, we establish complexity results that assess the difficulty of the problem. On the practical side, we design several new heuristics, which are shown to perform as efficiently as the best heuristics in [3, 2] although their cost is an order of magnitude lower.

J. Dongarra, D. Laforenza, S. Orlando (Eds.): Euro PVM/MPI 2003, LNCS 2840, pp. 657–660, 2003.

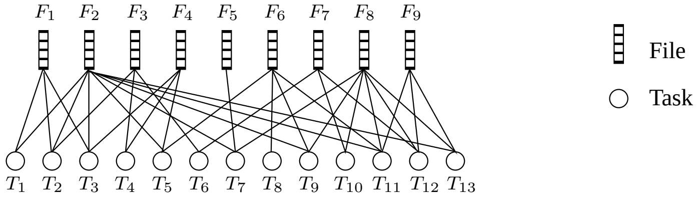

Fig. 1. Bipartite graph gathering the relations between the files and the tasks.

2 Framework

The problem is to schedule a set of n tasks $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Theses tasks have different sizes: the weight of task T_j is t_j, $1 \leq j \leq n$. There are no dependence constraints between the tasks, so they can be viewed as independent. However, the execution of each task depends upon one or several files, and a given file may be shared by several tasks. Altogether, there are m files in the set $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$. The size of file F_i is f_i, $1 \leq i \leq m$. We use a bipartite graph (see Figure 1 for an example) to represent the relations between files and tasks. Intuitively, a file F_i linked by an edge to a task T_j corresponds to some data that is needed for the execution of T_j to begin.

The tasks are scheduled and executed on a master-slave heterogeneous platform, with a master-processor P_0 and p slaves P_i, $1 \leq i \leq p$. Each slave P_q has a (relative) computing power w_q: it takes $t_j.w_q$ time-units to execute task T_j on processor P_q. The master processor P_0 initially holds all the m files in $\mathcal{F}$. The slaves are responsible for executing the n tasks in $\mathcal{T}$. Before it can execute a task T_j, a slave must have received from the master all the files that T_j depends upon. For communications, we use the one-port model: the master can only communicate with a single slave at a given time-step. We let c_q denote the inverse of the bandwidth of the link between P_0 and P_q, so that $f_i.c_q$ time-units are required to send file F_i from the master to slave P_q. We assume that communications can overlap computations on the slaves: a slave can process one task while receiving the files necessary for the execution of another task.

The objective is to minimize the total execution time. The schedule must decide which tasks will be executed by each slave, and it must determine the ordering in which the master sends the files to the slaves. Some files may well be sent several times, so that several slaves can independently process tasks that depend upon these files. Also, a file sent to some processor remains available for the rest of the schedule: if two tasks depending on the same file are scheduled on the same processor, the file must only be sent once.

3 Complexity

See the extended version [4] for a survey of existing results, and for the proof that the restricted instance of the problem with two slaves, and where all files and tasks have unit size ($t_j = f_i = 1$ for all j, i), remains NP-complete. Note that in that case, the heterogeneity only comes from the computing platform.

4 Heuristics

We compare our new heuristics to the three *reference heuristics* Min-min, Max-Min and Sufferage presented by Casanova et al. [3, 2]. All the reference heuristics are built on the following model: for each remaining task T_j, loop over all processors P_i and evaluate OBJECTIVE(T_j, P_i); pick the "best" task-processor pair (T_j, P_i) and schedule T_j on P_i as soon as possible. Here, OBJECTIVE(T_j, P_i) is the minimum completion time of task T_j if mapped on processor P_i, given the scheduling decisions that have already been made. The heuristics only differ by the definition of the "best" couple (T_j, P_i). For instance in Min-min, the "best" task T_j is the one minimizing the objective function when mapped on its most favorable processor: The computational complexity is at least $O(p.n^2 + p.n.|\mathcal{E}|)$, where $\mathcal{E}$ is the set of the edges in the bipartite graph.

When designing new heuristics, we took special care to decreasing the computational complexity. In order to avoid the loop on all the pairs of processors and tasks in the reference heuristics, we need to be able to pick (more or less) in constant time the next task to be scheduled. Thus we decided to sort the tasks *a priori* according to an objective function. However, since our platform is heterogeneous, the task characteristics may vary from one processor to the other. Therefore, we compute one sorted list of tasks for each processor. This sorted list is computed *a priori* and is not modified during the execution of the heuristic. Once the sorted lists are computed, we still have to map the tasks to the processors and to schedule them. The tasks are scheduled one-at-a-time. When we want to schedule a new task, on each processor P_i we evaluate the completion time of the first task (according to the sorted list) which has not yet been scheduled. Then we pick the pair task/processor with the lowest completion time. This way, we obtain an overall execution time reduced to $O(p.n.(\log n + |\mathcal{E}|))$.

Six different objective functions, and three refinement policies are described in [4], for a total of 48 variants. Here is a brief description of those that appear in Table 1 below:

- *Computation*: execution time of the task as if it was not depending on any file
- *Duration*: execution time of the task as if it was the only task to be scheduled on the platform
- *Payoff*: ratio of task duration over the sum of the sizes of its input files (when mapping a task, the time spent by the master to send the required files is payed by all the waiting processors, but the whole system gains the completion of the task)

The *readiness* refinement policy states to give priority to tasks whose input files are all already available at a given processor location, even though they are not ranked high in the priority list of that processor.

5 Experimental Results

Table 1 summarizes all the experiments. In this table, we report the best ten heuristics, together with their cost. This is a summary of $12,000$ random tests ($1,000$ tests over four graph types and three communication-to-computation cost ratios for the platforms, each with 20 heterogeneous processors and communication links). Each test involves 53 heuristics (5 reference heuristics and 48 combinations for our new heuristics). For each

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

Heuristic	Relative performance	Standard deviation	Relative cost	Standard deviation
Sufferage	1.110	0.1641	376.7	153.4
Min-min	1.130	0.1981	419.2	191.7
Computation+readiness	1.133	0.1097	1.569	0.4249
Computation+shared+readiness	1.133	0.1097	1.569	0.4249
Duration+locality+readiness	1.133	0.1295	1.499	0.4543
Duration+readiness	1.133	0.1299	1.446	0.3672
Payoff+shared+readiness	1.138	0.126	1.496	0.6052
Payoff+readiness	1.139	0.1266	1.246	0.2494
Payoff+shared+locality+readiness	1.145	0.1265	1.567	0.5765
Payoff+locality+readiness	1.145	0.1270	1.318	0.2329

test, we compute the ratio of the performance of all heuristics over the best heuristic. The best heuristic differs from test to test, which explains why no heuristic in Table 1 can achieve an average relative performance exactly equal to 1. In other words, the best heuristic is not always the best of each test, but it is closest to the best of each test in the average. The optimal relative performance of 1 would be achieved by picking, for any of the $12,000$ tests, the best heuristic for this particular case.

We see that *Sufferage* gives the best results: in average, it is within 11% of the optimal. The next nine heuristics closely follow: they are within 13% to 14.5% of the optimal. Out of these nine heuristics, only *Min-min* is a reference heuristic. In Table 1, we also report computational costs (CPU time needed by each heuristic). The theoretical analysis is confirmed: our new heuristics are an order of magnitude faster than the reference heuristics. We report more detailed performance data in [4]. As a conclusion, given their good performance compared to *Sufferage*, we believe that the eight new variants listed in Table 1 provide a very good alternative to the costly reference heuristics.

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. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Research Report 99-46, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, Sept. 1999.
3. 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.
4. 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. Available at `www.ens-lyon.fr/~yrobert`.
5. 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.