Computing and Informatics, Vol. 21, 2002, 413–430

STATIC SCHEDULING STRATEGIES FOR HETEROGENEOUS SYSTEMS

Olivier Beaumont

LaBRI, UMR CNRS 5800
Bordeaux, France
e-mail: Olivier.Beaumont@labri.fr

Arnaud Legrand, Yves Robert

LIP, UMR CNRS-INRIA 5668
ENS Lyon, France
e-mail: {Arnaud.Legrand;Yves.Robert}@ens-lyon.fr

Revised manuscript received 26 November 2002

Abstract. In this paper, we consider static scheduling techniques for heterogeneous systems, such as clusters and grids. We successively deal with minimum makespan scheduling, divisible load scheduling and steady-state scheduling. Finally, we discuss the limitations of static scheduling approaches.

Keywords: Static scheduling, one-port model, heuristics, divisible loads, steady-state scheduling, heterogeneous systems

1 INTRODUCTION

Scheduling computational tasks on a given set of processors is a key issue for high-performance computing. Although a large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous resources. However, future computing systems, such as the computational grid, are most likely to be widely distributed and strongly heterogeneous. In this paper, we consider the impact of heterogeneity on the design and analysis of static scheduling techniques:

how to enhance these techniques to efficiently address cluster and grid computing?

We begin with a brief review of scheduling heuristics designed to minimize the total schedule length, or makespan (Section 2). Next we sketch the divisible load approach in Section 3. We proceed to steady-state scheduling in Section 4. Finally, we discuss the limitations of static scheduling approaches in Section 5, and we state some concluding remarks in Section 6.

2 MINIMUM MAKESPAN SCHEDULING

2.1 Framework

The traditional objective of scheduling algorithms is the following: given a task graph and a set of computing resources, find a mapping of the tasks onto the processors, and order the execution of the tasks so that: (i) task precedence constraints are satisfied; (ii) resource constraints are satisfied; and (ii) a minimum schedule length is provided.

Task graph scheduling is usually studied using the so-called *macro-dataflow* model, which is widely used in the scheduling literature: see the survey papers [31, 36, 17, 21] and the references therein. This model was introduced for homogeneous processors, and has been (straightforwardly) extended for heterogeneous computing resources. In a word, there is a limited number of computing resources, or processors, to execute the tasks. Communication delays are taken into account as follows: let task T be a predecessor of task T' in the task graph; if both tasks are assigned to the same processor, no communication overhead is paid, the execution of T' can start right at the end of the execution of T; on the contrary, if T and T' are assigned to two different processors P_i and P_j, a communication delay is paid. More precisely, if P_i finishes the execution of T at time-step t, then P_j cannot start the execution of T' before time-step $t + \text{comm}(T, T', P_i, P_j)$, where $\text{comm}(T, T', P_i, P_j)$ is the communication delay, which depends upon both tasks T and T' and both processors P_i and P_j. Because memory accesses are typically one order of magnitude cheaper than inter-processor communications, it makes good sense to neglect them when T and T' are assigned to the same processor.

However, the major flaw of the macro-dataflow model is that communication resources are not limited. First, a processor can send (or receive) any number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Second, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously occur on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units.

2.2 Communication-Aware Models from the Literature

Communication-aware models restrict the use of communication links in various manners. In the model proposed by Sinnen and Sousa [39, 38, 40], the underlying communication network is no longer fully-connected. There are a limited number of communication links, and each processor is provided with a routing table which specifies the links to be used to communicate with another processor (hence the routing is fully static). The major modification is that at most one message can circulate on one link at a given time-step, so that contention for communication resources is taken into account.

Similarly, Hollermann et al. [25] and Hsu et al. [26] target networks of processors and introduce the following model: each processor can either send or receive a message at a given time-step (bidirectional communication is not possible); also, there is a fixed latency between the initiation of the communication by the sender and the beginning of the reception by the receiver. This model is rather close to the one-port model discussed below.

Several other papers impose restrictions on the communication resources, e.g. Tan et al. [42], Orduna et al. [33] and Roig et al. [34].

2.3 The One-Port Model

In this model, at a given time-step, any processor can communicate with at most another processor in both directions: sending to *and* receiving from another processor. The model also assumes communication/computation overlap. Note that several communications can occur in parallel, provided that they involve disjoint pairs of sending/receiving processors, which nicely models switches like Myrinet that can implement permutations [19], or even multiplexed bus architectures [27].

Several variants could be considered: no communication/computation overlap, uni-directional communications, or even a combination of both restrictions. But the full-overlap one-port model seems closer to the actual capabilities of modern processors, and we strongly advocate its use when targeting heterogeneous clusters.

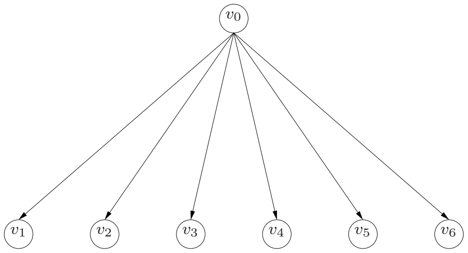


Fig. 1. Task graph for the example: all weights (nodes and communications) are equal to 1.

Serializing communications performed by the processors has a dramatic impact on the scheduling makespan. Consider the simple fork graph represented in Figure 1. Assume five same-speed processors and a fully homogeneous network, and suppose that task weights and communication costs are all equal to 1. In the macro-dataflow model, v_0 and the first two children v_1 and v_2 are assigned to processor P_0. One of the remaining children v_3, v_4, v_5 and v_6 is assigned to each remaining processor. P_0 executes task v_0 at time-step 0; then P_0 can perform all the four communications in parallel at time-step 1, while it executes v_1. The execution of v_2, v_3, v_4, v_5 and v_6 begins at time-step 2. The total makespan is equal to 3. In the one-port model, the same allocation of tasks to processors leads to a makespan at least 6: 1 for the parent task, 4 for the four messages to be sent sequentially, and 1 for the last task to be executed. One optimal solution is to assign three children tasks to P_0 and one remaining child task to a distinct processor (which makes one processor useless), for a makespan equal to 5. It is clear that communications from the parent node to the children has become the bottleneck. Of course, we could use larger task graphs and greater communication costs to come up with arbitrarily large differences in the makespans.

The one-port model turns out to be computationally even more difficult than the macro-dataflow model: scheduling a simple fork graph with an unlimited number of homogeneous processors is NP-hard [6]. Note that this problem has polynomial complexity in the macro-dataflow model [24]; we have to resort to fork-join graphs to get NP-completeness in the macro-dataflow model [17].

2.4 Heuristics

An impressive list of scheduling heuristics has been proposed in the literature for the macro-dataflow model with a limited number of homogeneous processors. More recently, several heuristics have been introduced to deal with different-speed processors, including the *minimum Partial Completion Time static priority* (PCT) heuristic [29], the *Best Imaginary Level* (BIL) heuristic [32], the *Critical Path on a Processor* (CPOP) heuristic [43], the *Generalized Dynamic Level* (GDL) heuristic [37] and the *Heterogeneous Earliest Finish Time* (HEFT) heuristic [43]. See [12, 13] for a survey and a comparison. Among these heuristics, HEFT is a natural extension of list-scheduling heuristics to cope with heterogeneous resources. More in particular, HEFT builds upon the old Modified Critical Path heuristic [23] and use bottom levels to assign priorities to tasks.

HEFT has been extended in [6] to fulfill the constraints of the one-port model. Furthermore, a new heuristic was introduced in [6], whose main characteristic is a better load-balancing at each decision step. This is achieved by considering a chunk of several ready tasks rather than a single one; the idea is to allocate to each processor a number of the tasks in the chunk whose overall processing time is proportional to its computing power.

Replacing the macro-dataflow by the one-port model is a first step towards designing realistic scheduling heuristics for heterogeneous clusters. However, such heuristics strongly depend upon an accurate knowledge of the whole task graph before execution, and they tend to require a precise estimation of the task and communication weights, which may limit their applicability to very regular problems arising from dense linear algebra, digital signal processing or multi-media applications.

3 DIVISIBLE LOAD SCHEDULING

3.1 Framework

The concept of divisible jobs has been introduced and widely studied by Robertazzi et al. [5, 41, 16, 11]. A divisible job is a job that can be arbitrarily split in a linear fashion among any number of processors. This corresponds to a perfectly parallel job: any sub-task can itself be processed in parallel, and on any number of processors. Such applications include the processing of large data files, Kalman filtering, and are a perfect testbed to understand the impact of realistic communication models, since the solution is trivial under the macro-dataflow model.

Robertazzi et al. studied the case of a bus (with homogeneous communication costs, heterogeneous computation costs and at most one communication at a given time step on the bus) in [41], the case of a tree of processors (with homogeneous communication and computation costs, using the one-port model) in [5], and the case of a star (heterogeneous communication and computation costs, one-port model) in [16]. In this section, we present their main results for bus and star architectures.

The notations used through this section are the following:

- α_i denotes the fraction of workload assigned to processor P_i, $\forall i$ ($\sum_i \alpha_i = 1$).
- w_i denotes the inverse of the processing speed of processor P_i, normalized so that $\alpha_i w_i$ denotes the time required by P_i to process its load fraction.
- c_i denotes the inverse of the communicating speed between processor P_i and the originating processor, normalized so that $\alpha_i c_i$ denotes the time required to transmit to P_i its load fraction. In the case of a bus, $c_i = c$, $\forall i$.
- T_i denotes the time elapsed before P_i begins its processing. Thus, $T_f = \max_i(T_i + \alpha_i w_i)$ denotes the overall computational time.

3.2 Case of a Bus

In general, two main problems are to be solved for dispatching divisible jobs. The first problem is to determine in which order the work should be sent to the different processors. Since the bus communication medium can handle only one communication at a given time step, the solution is as depicted in Figure 2. Once the communication order has been determined, the second problem is to decide how much work should be allocated to each processor P_i. The final objective is to minimize the makespan.

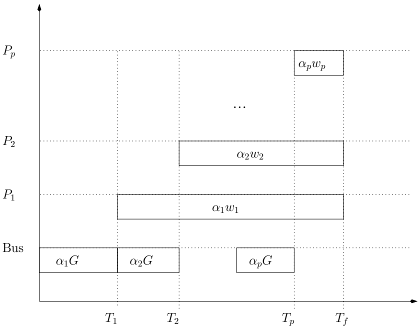


Fig. 2. Pattern of a solution for dispatching the load of a divisible job

In the case of a bus, the solution is surprisingly simple. First, one can prove that all the processors must finish their work at the same time (i.e. $T_i + \alpha_i w_i = T_f, \ \forall i$). Indeed, otherwise, some work could be transferred from a busy processor to an idle one in order to reduce T_f. Thus, the following system of equation holds,

$$\begin{cases} T_f - T_i = \alpha_i w_i, & \forall 1 \leq i \leq n \\ T_{i+1} - T_i = \alpha_{i+1} c & \forall 1 \leq i \leq n-1 \end{cases}$$

if data is sent successively to $P_2, \ldots, P_n$. Closed forms can be obtained for both the α_i's and T_f. These closed forms are rather complicated, although the method for obtaining them is elementary, and we refer the reader to [41] to find the actual algebraic expressions. The surprising and interesting point is that the overall computational time T_f does not depend upon the order chosen for sending data to the different processors, so that the ordering $P_2, \ldots, P_n$ is in fact optimal.

Therefore, closed forms for the optimal solution can be derived when the communication medium is a bus.

3.3 Case of a Star

The case of an heterogeneous star is discussed in [16]. The solution can again be depicted as in Figure 2, with proper c_i values for each processor P_i (so that $\alpha_i c$ is changed into $\alpha_i c_i$). The results are less satisfying than in the case of the bus. Indeed, the main known result is that if data is sent to the different processors in a given order (say, again, $P_2, \ldots, P_n$), then closed forms can be obtained for both the α_i's and T_f. Unfortunately, the formal proof of the result stating that all the processors must finish their work at the same time does not hold in the heterogeneous case. Moreover, T_f strongly depends on the communication ordering, and to the best of our knowledge, the optimal communication ordering is not known.

As a conclusion, we point out that even with this very simple application model, on a very common architecture (a star of heterogeneous processors), deriving optimal solutions is very difficult. One interesting problem would consist in considering both initial and back communications, so as to model an application where results have to be sent back to the originating processor. The problem is open, but we expect it to be challenging in this case, since two permutations (for initial and back communications) are to be determined.

3.4 Multi-Round Algorithms

The master processor can distribute the chunks to the workers in a single round, as above, so that there will be a single communication between the master and each worker. This is the simplest situation, but for large workloads, the single round approach is not efficient, because of the idle time incurred by the last processors to receive their chunks. To minimize the makespan, i.e. the total execution time, the master will send the chunks to the workers in multiple rounds: the communications will be shorter (less latency) and pipelined, and the workers will be able to compute the current chunk while receiving data for the next one.

To remain realistic, the model must be enhanced, and communication latencies should be included (otherwise the optimal strategy would be to use an infinite number of rounds). For each communication of size L between the master and a worker, say P_i, we pay a latency g_i and a linear term $L.c_i$. Latencies play an important role in current architectures [19], and have been introduced for divisible loads by Drozdowski [20]. They now are widely used in the literature.

Deriving an efficient multi-round solution becomes a challenging problem: how many rounds should be scheduled? what is the best size of the chunks for each round? Intuitively, the size of the chunks should be small in the first rounds, so as to start all the workers as soon as possible. Then the chunk size should increase to a steady-state value, to be determined so as to optimize the usage of the total available bandwidth of the network. Finally the chunk size should be decreased while reaching the end of the computation. In Chapter 10 of [11], there is no quantified value provided for the number of rounds to be used. Recently, Altilar and Paker [1, 2], and Yang and Casanova [45] have introduced multi-round algorithms and analytically expressed their performance. We refer the reader to [7] for a bibliographical survey of multi-round algorithms, and for the design of an asymptotically optimal algorithm.

4 STEADY-STATE SCHEDULING

In this section we deal with large problems. In this context an absolute minimization of the total execution time is not really required. Indeed, deriving asymptotically optimal schedules is more than enough to ensure an efficient use of the architectural resources. In a word, the idea to reach asymptotic optimality is to relax the problem: (i) neglect the initialization and clean-up phases, and concentrate on steady-state operation; (ii) derive an optimal steady-state scheduling using linear programming; (iii) prove the asymptotic optimality of the associated schedule. We give two examples of this approach below: packet routing, and mixed task/data parallelism.

4.1 Packet Routing

The packet routing problem is the following: let $G = (V, E)$ be a non-oriented graph modeling the target architectural platform, and consider a set of same-size packets to be routed through the network. Each packet is characterized by a source node (where it initially resides) and a destination node (where it must be located in the end). For each pair of nodes (v_k, v_l) in G, let n_{kl} be the number of packets to be routed from v_k to v_l. Let

$$\mathcal{P} = \{(k,l) \in V^2, \quad n_{kl} \neq 0\}.$$

Bertsimas and Gamarnik [10] introduce a scheduling algorithm which is asymptotically optimal when $n = \sum_{(k,l)\in\mathcal{P}} n_{kl} \to +\infty$. So to speak, temporal constraints have been removed in this algorithm: it is never written than a packet must have reached a node before leaving it.

Consider an arbitrary scheduling and let x_{ij}^{kl} be the number of packets circulating from v_k to v_l and using the edge (the communication link) between v_i and v_j, $\forall (i,j) \in E,\ \forall (k,l) \in \mathcal{P}$. The time needed to circulate a packet on any edge is assumed to be constant (equal to 1), but at most one packet can circulate on one edge at a given time-step. We obtain the following *relaxed* linear program:

$$\begin{array}{l}
\textsc{Minimize } C_{\max}, \\
\textsc{subject to} \\
\left\{\begin{array}{ll}
(1)\ \sum_{i,(k,i)\in E} x_{ki}^{kl} = n_{kl} & \forall (k,l) \in \mathcal{P} \\
(2)\ \sum_{i,(i,l)\in E} x_{il}^{kl} = n_{kl} & \forall (k,l) \in \mathcal{P} \\
(3)\ \sum_{j,(j,i)\in E} x_{ji}^{kl} = \sum_{r,(i,r)\in E} x_{ir}^{kl} & \forall (k,l) \in \mathcal{P},\ \forall i \neq k,l \\
(4)\ C_{i,j} = \sum_{(k,l)\in\mathcal{P}} x_{ij}^{kl} & \forall (i,j) \in E \\
(5)\ C_{i,j} \leq C_{\max}, & \forall (i,j) \in E \\
(6)\ x_{ij}^{kl} \geq 0, \quad C_{i,j} \geq 0, & \forall (k,l) \in \mathcal{P},\ (i,j) \in E
\end{array}\right.
\end{array}$$

The first two equations state that the number of packets of type (k,l) that leave node k and reach node l is n_{kl}. Equation (3) is the conservation law (conservation of the number of packets) at node i. Equation (4) defines the total occupation time of edge (i,j), and equation (5) states that all these occupation times minor the makespan $C_{\max}$. Note that all temporal constraints have been left out, hence the name *relaxed*.

The solution of this linear program with $O(|E||P|)$ rational variables and $O(|V||P| + |E|)$ constraints can be obtained in polynomial time. The complexity does not depend on n, the total number of packets, which justifies its use when n is large. Now, to construct the actual scheduling, we split the execution into phases, and we reproduce a "rounded" version of the relaxed solution during each phases. Let Ω be the length of a phase (to be determined later) and let

$$a_{ij}^{kl} = \left\lfloor \frac{x_{ij}^{kl}\Omega}{C_{\max}} \right\rfloor, \quad \forall (k,l) \in \mathcal{P},\ (i,j) \in E$$

be the number of packets (rounded from below) of type (k, l) which circulate on the edge (i, j) during Ω time-steps in the relaxed problem. The algorithm proposed in [10] is the following:

Input Compute the optimal value $C_{\max}$ from the relaxed linear program.

Step 1 During each phase $[l\Omega, (l + 1)\Omega]$, where $l = 0, \ldots, \lceil \frac{C_{\max}}{\Omega} \rceil - 1$, and for each edge $(i, j) \in E$, circulate on the edge as many packets of type (k, l) as available in node i at time $l\Omega$, but no more than a_{ij}^{kl}.

Step 2 At time-step $T = \lceil \frac{C_{\max}}{\Omega} \rceil \Omega$, all the packets that have not been fully routed are handled sequentially.

It can be proven that at time-step

$$\left(\frac{C_{\max}}{\Omega} + 1\right) \Omega + |E||V| \left(\frac{C_{\max}|P|}{\Omega} + |P| + \Omega\right)$$

all the packets have successfully been routed. The proof sketch is as follows. First the previous scheduling is shown feasible (during each phase, all the packets can indeed be transmitted). Next, at the end of Step 1, whose length is not larger than $(\frac{C_{\max}}{\Omega}+1)\Omega$, the number of packets that have not reached their destination is bounded by $|E|(\frac{C_{\max}|P|}{\Omega}+|P|+\Omega)$. These packets are routed sequentially on a path of length at most $|V|$, hence the duration of Step 2 is not larger than $|E||V|(\frac{C_{\max}|P|}{\Omega}+|P|+\Omega)$. If we choose Ω of the order of $\sqrt{C_{\max}}$, the makespan of the schedule is $C_{\max}+O(\sqrt{C_{\max}})$, hence the asymptotic optimality.

4.2 Mixed Task/Data Parallelism

We consider here applications that consist of a suite of identical, independent problems to be solved. In turn, each problem consists of a set of tasks, with dependences between these tasks. A typical example is the repeated execution of the same algorithm on several distinct data samples: the task graph of the algorithm is executed several times, one for each problem instance. The application is executed using the master-slave paradigm: one particular processor holds (or produces) all the data that is initially needed. Tasks (or more precisely data files associated to them) are distributed to, and executed by, the other processors (the slaves). Note that different copies of the same task type (corresponding to different problem instances) may well be executed by different processors.

The objective is to derive an efficient scheme for the distribution and the scheduling of the tasks to the processors. We use the following notations (see Figure 3):

- The task graph is $G = (T, C)$. Each vertex T_k represents a task type to be executed, and each edge $(T_k \rightarrow T_l)$ represents a communication between two tasks, and is weighted by $data_{k,l}$, the volume of communication to be exchanged (think of each edge as been associated to a file of type (k, l) to by sent from T_k to T_l).

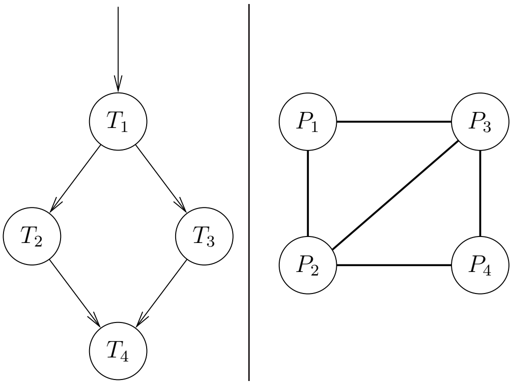


Fig. 3. The application/architecture framework

- The platform graph is $G' = (P, L)$. Vertices represent computing resources and edges represent communication links. Each edge in L is weighted $c_{i,j}$, the time needed to transfer one data unit on the link from P_i to P_j.
- The time needed to execute (any copy of) task T_k on processor P_i is $w_{i,k}$. The time needed to communicate one file of type (k, l) (related to the edge from T_k to T_l in the task graph) along the communication link from P_i to P_j in the platform graph is $c_{i,j} \times data_{k,l}$.
- We use the full-overlap one-port model of Section 2.3: at a given time-step, a processor can simultaneously execute a task, receive a message (at most one) and send a message (at most one).

This model is quite general, and deriving a minimum makespan schedule is hopeless. As in Section 4.1, we introduce a relaxed problem, which characterizes the optimal steady-state operation, i.e. the maximal throughput (total number of tasks executed per time-unit). We use the following notations:

- $s(P_i, P_j, T_k, T_l)$ is the fraction of time spent each time-unit by P_i to send to P_j data involved by the edge (T_k, T_l) of the task graph. Similarly, $\mathrm{Sent}(P_i, P_j, T_k, T_l)$ is the number of data files of this type sent along the edge (P_i, P_j) per time-unit, with $s(P_i, P_j, T_k, T_l) = \mathrm{Sent}(P_i, P_j, T_k, T_l) * data_{k,l} * c_{i,j}$.
- $\alpha(P_i, T_k)$ is the fraction of time spent each time-unit by P_i to compute tasks of type T_k. Similarly, $\mathrm{Cons}(P_i, T_k)$ is the number of tasks of this type consumed by P_i each time-unit, with $\alpha(P_i, T_k) = \mathrm{Cons}(P_i, T_k) * wi, k$.
- Finally, we add two fictitious tasks T_{begin} and T_{end} to the task graph. T_{begin} is the predecessor of all input tasks in G (tasks without any predecessor in G). The execution time of T_{begin} by any processor is equal to 0, and the communication volume along any edge from T_{begin} to an input task is also 0. Similarly, T_{end} is the successor of all output tasks in G.

Let P_{ms} be the master processor, and let $n(P_i)$ be the set of the neighbors of P_i in the platform graph. The following linear program summarizes the equations governing the activity of the processors and of the communication links within one time-unit, as well as conservation laws for each task file and each data file type:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textsc{Maximize} \ \sum_i \text{Cons}(P_i, T_{\text{end}}),} \\
\multicolumn{2}{l}{\textsc{subject to}} \\
(1)\ \forall i, \forall k, & 0 \leq \alpha(P_i, T_k) \leq 1 \\
(2)\ \forall i, j, k, l, & 0 \leq s(P_i, P_j, T_k, T_l) \leq 1 \\
(3)\ \forall i, j, k, l, & s(P_i, P_j, T_k, T_l) = \text{Sent}(P_i, P_j, T_k, T_l) * \mathit{data}_{k,l} * c_{i,j} \\
(4)\ \forall i, k, & \alpha(P_i, T_k) = \text{Cons}(P_i, T_k) * wi, k \\
(5)\ \forall i, & \sum_{P_j \in n(P_i)} \sum_{(k,l) \in C} s(P_i, P_j, T_k, T_l) \leq 1 \\
(6)\ \forall i, & \sum_{P_j \in n(P_i)} \sum_{(k,l) \in C} s(P_j, P_i, T_k, T_l) \leq 1 \\
(7)\ \forall i, & \sum_{T_k \in T} \alpha(P_i, T_k) \leq 1 \\
(8)\ \forall i, & \text{Cons}(P_i, T_{\text{begin}}) = 0 \\
(9)\ \forall i, j, k, & s(P_i, P_j, T_k, T_{\text{END}}) = 0 \\
(10)\ \forall i, k, l, & \sum_{P_j \in n(P_i)} \text{Sent}(P_j, P_i, T_k, T_l) + \text{Cons}(P_i, T_k) = \\
 & \quad \sum_{P_j \in n(P_i)} \text{Sent}(P_i, P_j, T_k, T_l) + \text{Cons}(P_i, T_l) \\
(11)\ \forall i, k \neq \text{begin}, l, & \sum_{P_j \in n(P_{\text{ms}})} \text{Sent}(P_j, P_{\text{ms}}, T_k, T_l) + \text{Cons}(P_{\text{ms}}, T_k) = \\
 & \quad \sum_{P_j \in n(P_{\text{ms}})} (\text{Sent}(P_{\text{ms}}, P_j, T_k, T_l) + \text{Cons}(P_{\text{ms}}, T_l)
\end{array}
$$

The objective function is equal to the number of copies of task T_{end} executed per time-step. Because of the dependences, the availability of a copy of T_{end} means that the whole task graph instance has been executed. Equation (5) states that the fraction of time spent by P_i to send tasks cannot exceed 1; sending is sequential in the one-port model, hence the summation on the neighbors. Equation (6) is the counterpart for receptions, as well as equation (7) for computations. Equation (10), and its variant equation (11) for the master processor, is the most important: consider a given processor P_i, and a given edge (T_k, T_l) in the task graph. During each time unit, P_i receives from its neighbors a given number of files of type (T_k, T_l). Processor P_i itself executes some tasks T_k, thereby generating as many new files of type (T_k, T_l). What does happen to these files? Some are sent to the neighbors of P_i, and some are consumed by P_i to execute tasks of type T_l: we derive equation (10), which really applies to the steady-state operation. At the beginning of the operation of the platform, only input tasks are available to be forwarded. Then some computations take place, and tasks of other types are generated. At the end of this initialization phase, we enter the steady-state: during each time-period in steady-state, each processor can simultaneously perform some computations, and send/receive some other tasks. This is why equation (10) is sufficient, we do not have to detail which operation is performed at which time-step.

Finally, we have derived a linear program whose complexity is polynomial in $|T|$, $|C|$, $|P|$ and $|L|$, and does not depend upon the number of problems (task graphs) to deal with. In this case, deriving a practical scheduling is easier than in Section 4.1.

Having computed the solution of the linear program, we derive the time period T by computing the least common multiple of all denominators of the rational variables: we obtain an interval of length T during which the number of tasks executed and transmitted is an integer constant. Using a sequential initialization phase to feed the processors, and a sequential clean-up phase to process the very last tasks, we derive an asymptotically optimal schedule. More precisely, the number of tasks executed by this schedule is optimal, up to a constant that only depends upon the task graph and platform graph, not upon the total number of tasks. See [4, 8] for further details.

5 LIMITATIONS OF STATIC SCHEDULING

We have surveyed three useful techniques when targeting heterogeneous clusters:

- Replacing the macro-dataflow model by the one-port model is a first step towards designing realistic scheduling heuristics.
- Assuming a perfectly divisible load greatly simplifies the task allocation problem.
- Dealing with steady-state operation instead of makespan minimization is a nice way to circumvent the computational complexity of scheduling problems while deriving efficient (often asympotically optimal) scheduling algorithms.

However, several problems remain to be addressed. We classify them into the following two categories: acquiring a good knowledge of the platform graph, and running extensive experiments or simulations.

5.1 Knowledge of the Platform Graph

Is it realistic to assume that all the information concerning the task graph is available from the very beginning of the scheduling? For some applications, tasks are only known *on-line*, as the computation progresses. But there are regular problems (e.g. a two-dimensional FFT, or a dense LU solver) for which the whole division into tasks, and the dependences between the tasks, is known a priori. For such problems, the *structure* of the task graph (nodes and edges) only depends upon the application, not upon the target platform. Problems arise from the weights, i.e. the estimation of the execution times and of the communication times. For instance, critical path scheduling relies on a precise knowledge of all these parameters to assign the next ready task to the adequate computing resource. Even the steady-state scheduling of independent tasks requires some static knowledge of the architecture.

A classical answer to this problem is borrowed from a simple paradigm used in dynamic strategies, namely *"use the past to predict the future"*, i.e. use the currently observed speed of computation of each machine and of each communication link to decide for the next distribution of work [9]. There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation [3, 18]. The situation is even worse for communication links, because of unpredictable contention problems.

When deploying an application on a platform, the idea is thus to divide the scheduling into phases. During each phase, all the machine and network parameters are collected and histogrammed, using a tool like NWS [44]. This information will then guide the scheduling decisions for the next phase.

Moving from heterogeneous clusters to computational grids will cause further problems. Even discovering the characteristics of the surrounding computing resources may prove a difficult task, despite the availability of tools like IDMaps and Global Network Positioning [22, 30] or Effective Network View [35]. Still, even in the favorable case where the target platform graph has been well identified and is relatively stable, schedulers face two major difficulties: (i) providing an accurate modeling of the hierarchical structure of the platform and (ii) designing scheduling algorithms that are well-suited to this hierarchical structure. Overcoming these two difficulties will be a challenging task for the forthcoming years.

5.2 Experiments Versus Simulations

Real experiments on the target platform are often involved to test or to compare heuristics. However, on a distributed heterogeneous platform, such experiments are technically difficult to drive, because of the genuine instability of the platform. For example, wide-area links are often shared with Internet traffic from other applications, and their performance is not as constant and reliable as the one of a dedicated cluster of workstations. In a word, it is almost impossible to guarantee that a platform which is not dedicated to the experiment, will remain exactly the same between two tests, thereby forbidding any meaningful comparison.

Simulations are then used to replace real experiments, so as to ensure the reproducibility of measured data. Being faster than real experiments, simulations will enable to test the algorithms in a variety of conditions. A key issue is the possibility to run the simulations against a realistic environment. The main idea of trace-based scheduling is to record the platform parameters today, and to simulate the algorithms tomorrow, against the recorded data: even though it is not the current load of the platform, it is realistic, because it represents a fair summary of what happened previously.

A good example of a trace-based simulation tool is SimGrid [14], a toolkit providing a set of core abstractions and functionalities that can be used to easily build simulators for specific application domains and/or computing environment topologies. SimGrid performs event-driven simulation. The most important component of the simulation process is the resource modeling. The current implementation assumes that resources have two performance characteristics: latency (time in seconds to access the resource) and service rate (number of work units performed per time unit). SimGrid provides mechanisms to model performance characteristics either as constants or from traces. This means that the latency and service rate of each resource can be modeled by a vector of time-stamped values, or trace. Traces allow the simulation of arbitrary performance fluctuations such as the ones observable for real resources. In essence, traces are used to account for potential background load on resources that are time-shared with other applications/users. SimGrid has been successfully used to evaluate scheduling strategies for parameter sweep applications over the computational grid [15]. An extension of SimGrid to decentralized schedulers and realistic platforms is currently under development [28].

6 CONCLUSION

The difficulty of scheduling for clusters and grids should not be underestimated. Data decomposition, task allocation and load balancing were known to be difficult problems in the context of classical parallel architectures. They become extremely difficult in the context of heterogeneous clusters, not to mention grid computing platforms. If the platform is not stable enough, or if it evolves too fast, dynamic schedulers are the only option. Otherwise, there is always the opportunity to inject some static knowledge into dynamic schedulers. Future work will decide whether this opportunity is a niche (the pessimistic answer) or whether it encompasses a wide range of applications (the expected answer!).

REFERENCES

[1] Altilar, D.—Paker, Y.: An Optimal Scheduling Algorithm for Parallel Video Processing. In IEEE Int. Conference on Multimedia Computing and Systems, IEEE Computer Society Press, 1998.

[2] Altilar, D.—Paker, Y.: Optimal Scheduling Algorithms for Communication Constrained Parallel Processing. In Euro-Par 2002, LNCS 2400, pp. 197–206, Springer Verlag, 2002.

[3] Anastasiadis, S.—Sevcik, K. C.: Parallel Application Scheduling on Networsk of Workstations. Journal of Parallel and Distributed Computing, Vol. 43, 1997, pp. 109–124.

[4] Banino, C.—Beaumont, O.—Legrand, A.—Robert, Y.: Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids. In PARA'02: International Conference on Applied Parallel Computing, LNCS 2367, pp. 423–432, Springer Verlag, 2002.

[5] Bataineh, S.—Hsiung, T. Y.—Robertazzi, T. G.: Closed form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job. IEEE Transactions on Computers, Vol. 43, 1994, No. 10, pp. 1184–1196.

[6] Beaumont, O.—Boudet, V.—Robert, Y.: A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors. In HCW'2002, the 11th Heterogeneous Computing Workshop. IEEE Computer Society Press, 2002.

[7] Beaumont, O.—Legrand, A.—Robert, Y.: Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems. Technical Report 2002-36, LIP, ENS Lyon, France, October 2002.

[8] Beaumont, O.—Legrand, A.—Robert, Y.: Scheduling Strategies for Mixed Data and Task Parallelism on Heterogeneous Clusters and Grids. In PDP'2003, 11th Euromicro Workshop on Parallel, Distributed and Network-based Processing, IEEE Computer Society Press, 2003.

[9] Berman, F.: High-Performance Schedulers. In I. Foster and C. Kesselman, editors, The Grid: Blueprint for a New Computing Infrastructure, pp. 279–309, Morgan-Kaufmann, 1999.

[10] Bertsimas, D.—Gamarnik, D.: Asymptotically Optimal Algorithm for Job Shop Scheduling and Packet Routing. Journal of Algorithms, Vol. 33, 1999, No. 2, pp. 296–318.

[11] Bharadwaj, V.—Ghose, D.—Mani, V.—Robertazzi, T. G.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press, 1996.

[12] Braun, T. D.—Siegel, H. J.—Beck, N.—Blni, L. L.—Maheswaran, M.—Reuther, A.—Robertson, J. P.—Theys, M. D.—Yao, B.—Hensgen, D.—Freund, R. F.: A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems. In Eight Heterogeneous Computing Workshop, pp. 15–29, IEEE Computer Society Press, 1999.

[13] Braun, T. D.—Siegel, H. J.—Maciejewski, A. A.: Static Mapping Heuristics for Tasks with Dependencies, Priorities, Deadlines, and Multiple Versions in Heterogeneous Environments. In International Parallel and Distributed Processing Symposium (IPDPS'2002), IEEE Computer Society Press, 2002.

[14] Casanova, H.: Simgrid: A Toolkit for the Simulation of Application Scheduling. In Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CC-Grid'01), IEEE Computer Society, May 2001.

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

[16] Charcranoon, S.—Robertazzi, T. G.—Luryi, S.: Optimizing Computing Costs Using Divisible Load Analysis. IEEE Transactions on Computers, Vol. 49, 2000, No. 9, pp. 987–991.

[17] Chrétienne, P.—Coffman, E. G. Jr.—Lenstra, J. K.—Liu, Z. editors: Scheduling Theory and Its Applications. John Wiley and Sons, 1995.

[18] Cierniak, M.—Zaki, M. J.—Li, W.: Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations. The Computer Journal, Vol. 40, 1997, No. 6, pp. 356–372.

[19] Culler, D. E.—Singh, J. P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, San Francisco, CA, 1999.

[20] Drozdowski, M.: Selected Problems of Scheduling Tasks in Multiprocessor Computing Systems. PhD thesis, Instytut Informatyki Politechnika Poznanska, Poznan, 1997.

[21] El-Rewini, H.—Ali, H. H.—Lewis, T. G.: Task Scheduling in Multiprocessing Systems. Computer, Vol. 28, 1995, No. 12, pp. 27–37.

[22] Francis, P.—Jamin, S.—Jin, C.—Jin, Y.—Raz, D.—Shavitt, Y.—Zhang, L.: Idmaps: A Global Internet Host Distance Estimation Service. IEEE/ACM Transactions on Networking, October 2001.

[23] Gerasoulis, A.—Yang, T.: A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. J. Parallel and Distributed computing, Vol. 16, 1992, No. 4, pp. 276–291.

[24] Gerasoulis, A.—Yang, T.: On the Granularity and Clustering of Directed Acyclic Task Graphs. IEEE Trans. Parallel and Distributed Systems, Vol. 4, 1993, No. 6, pp. 686–701.

[25] Hollermann, L.—Hsu, T. S.—Lopez, D. R.—Vertanen, K.: Scheduling Problems in a Practical Allocation Model. J. Combinatorial Optimization, Vol. 1, 1997, No. 2, pp. 129–149.

[26] Hsu, T. S.—Lee, J. C.—Lopez, D. R.—Royce, W. A.: Task Allocation on a Network of Processors. IEEE Trans. Computers, Vol. 49, 2000, No. 12, pp. 1339–1353.

[27] Hwang, K.—Xu, Z.: Scalable Parallel Computing. McGraw-Hill, 1998.

[28] Lerouge, J.—Legrand, A.: Towards Realistic Scheduling Simulation of Distributed Applications. Technical Report 2002-28, LIP, ENS Lyon, France, July 2002.

[29] Maheswaran, M.—Siegel, H. J.: A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. In Seventh Heterogeneous Computing Workshop. IEEE Computer Society Press, 1998.

[30] Ng, E.—Zhang, H.: Predicting Internet Network Distance with Coordinates-Based Approaches. In InfoCom'O2, IEEE Computer Society Press, 2002.

[31] Norman, M. G.—Thanisch, P.: Models of Machines and Computation for Mapping in Multicomputers. ACM Computing Surveys, Vol. 25, 1993, No. 3, pp. 103–117.

[32] Oh, H.—Ha, S.: A Static Scheduling Heuristic for Heterogeneous Processors. In Proceedings of Europar'96, LNCS 1123, Springer Verlag, 1996.

[33] Orduna, J. M.—Silla, F.—Duato, J.: A New Task Mapping Technique for Communication-Aware Scheduling Strategies. In T. M. Pinkston, editor, Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01), pp. 349–354, IEEE Computer Society Press, 2001.

[34] Roig, C.—Ripoll, A.—Senar, M. A.—Guirado, F.—Luque, E.: Improving Static Scheduling Using Inter-Task Concurrency Measures. In T. M. Pinkston, editor, Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01), pp. 375–381, IEEE Computer Society Press, 2001.

[35] Shao, G.—Berman, F.—Wolski, R.: Using Effective Network Views to Promote Distributed Application Performance. In International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, June 1999.

[36] Shirazi, B. A.—Hurson, A. R.—Kavi, K. M.: Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Science Press, 1995.

[37] Sih, G. C.—Lee, E. A.: A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, 1993, No. 2, pp. 175–187.

[38] Sinnen, O.—Sousa, L.: Comparison of Contention-Aware List Scheduling Heuristics for Cluster Computing. In T. M. Pinkston, editor, Workshop for Scheduling and

Resource Management for Cluster Computing (ICPP'01), pp. 382–387, IEEE Computer Society Press, 2001.

[39] Sinnen, O.—Sousa, L.: Exploiting Unused Time-Slots in List Scheduling Considering Communication Contention. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, EuroPar'2001 Parallel Processing, pp. 166–170, Springer-Verlag LNCS 2150, 2001.

[40] Sinnen, O.—Sousa, L.: Scheduling Task Graphs on Arbitrary Processor Architectures Considering Contention. In High Performance Computing and Networking, pp. 373–382, Springer-Verlag LNCS 2110, 2001.

[41] Sohn, J.—Robertazzi, T. G.—Luryi, S.: Optimizing Computing Costs Using Divisible Load Analysis. IEEE Transactions on Parallel and Distributed Systems, Vol. 9, 1998, No. 3, pp. 225–234.

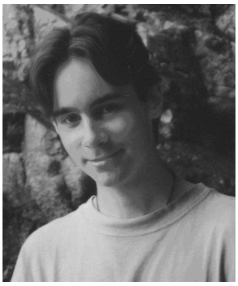
[42] Tan, M.—Siegel, H. J.—Antonio, J. K.—Li, Y. A.: Minimizing the Aplication Execution Time Through Scheduling of Subtasks and Communication Traffic in a Heterogeneous Computing System. IEEE Transactions on Parallel and Distributed Systems, Vol. 8, 1997, No. 8, pp. 857–871.

[43] Topcuoglu, H.—Hariri, S.—Wu, M.-Y.: Task Scheduling Algorithms for Heterogeneous Processors. In Eighth Heterogeneous Computing Workshop, IEEE Computer Society Press, 1999.

[44] Wolski, R.—Spring, N. T.—Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Future Generation Computer Systems, Vol. 15, 1999, No. 10, pp. 757–768.

[45] Yang, Y.—Casanova, H.: Multi-Round Algorithm for Scheduling Divisible Workload Applications: Analysis and Experimental Evaluation. Technical Report CS2002-0721, Dept. of Computer Science and Engineering, University of California, San Diego, 2002.

Olivier Beaumont was born in 1970. He obtained a PhD degree from the Université de Rennes in January 1999. He is currently an associate professor in the LaBRI laboratory in Bordeaux. His main research interests are parallel algorithms on distributed memory architectures.



Arnaud Legrand was born in 1977. He is currently a PhD student in the Computer Science Laboratory LIP at ENS Lyon. He is mainly interested in parallel algorithm design for heterogeneous platforms and in scheduling.

Yves Robert was born in 1958. He obtained a PhD degree from Institut National Polytechnique de Grenoble in January 1986. He is currently a full professor in the Computer Science Laboratory LIP at ENS Lyon. He is the author of four books, 80 papers published in international journals, and 100 papers published in international conferences. His main research interests include scheduling techniques and parallel algorithms for clusters and grids. He is a member of ACM and IEEE, and serves as an associate editor of IEEE TPDS.