Static scheduling strategies for heterogeneous systems
Olivier Beaumont, Arnaud Legrand, Yves Robert

To cite this version:

HAL Id: hal-02101858
https://hal-lara.archives-ouvertes.fr/hal-02101858
Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Static scheduling strategies for heterogeneous systems

O. Beaumont, A. Legrand and Y. Robert

July 2002

Research Report N° 2002-29
Static scheduling strategies for heterogeneous systems

O. Beaumont, A. Legrand and Y. Robert

July 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, clusters, grids, makespan, divisible load, steady-state

Résumé
Nous faisons un tour d’horizon des techniques d’ordonnancement statiques pour plateformes hétérogènes, comme les grappes et les grilles de calcul. Nous nous intéressons successivement aux heuristiques pour la minimisation du temps d’exécution total, aux méthodes d’allocation de tâches infiniment divisibles, et à l’obtention du meilleur régime permanent pour des problèmes de grande taille. Enfin, nous discutons brièvement des limitations de toutes ces techniques statiques.

Mots-clés: ordonnancement, grappes de calcul, méthodes statiques
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 2.1). 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 [NT93, SHK95, CML95] 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 interprocessor 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 Sinmen and Sousa, 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 an other 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. and Hsu et al. 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. [TSAL97], Orduna et al. [OSD01] and Roig et al. [RRS+01].

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, or even multiplexed bus architectures.

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.

![Figure 1: Task graph for the example: all weights (nodes and communications) are equal to 1.](image)

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, v₀ and the first two children v₁ and v₂ are assigned to processor P₀. One of the remaining children v₃, v₄, v₅ and v₆ is assigned to each remaining processor. P₀ executes task v₀ at time-step 0; then P₀ can perform all the four communications in parallel at time-step 1. The total makespan is
then 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 \( \text{[BBR02]} \). Note that this problem has polynomial complexity in the macro-dataflow model \( \text{[GY92]} \); we have to resort to fork-join graphs to get NP-completeness in the macro-dataflow model \( \text{[CJLL95]} \).

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 \textit{minimum Partial Completion Time static priority} (PCT) heuristic \( \text{[MS98]} \), the \textit{Best Imaginary Level} (BIL) heuristic \( \text{[OH96]} \), the \textit{Critical Path on a Processor} (CPOP) heuristic \( \text{[THW99]} \), the \textit{Generalized Dynamic Level} (GDL) heuristic \( \text{[SL93]} \) and the \textit{Heterogeneous Earliest Finish Time} (HEFT) heuristic \( \text{[THW99]} \). 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 \textit{Modified Critical Path} heuristic \( \text{[CY92]} \) and use bottom levels to assign priorities to tasks.

HEFT has been extended in \( \text{[BBR02]} \) to fulfill the constraints of the one-port model. Furthermore, a new heuristic was introduced in \( \text{[BBR02]} \), 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. \( \text{[BHR94]} \) \( \text{[SL98]} \) \( \text{[CRL00]} \) \( \text{[BGM96]} \). 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
tested 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 [SRL98], the case of a tree of processors (with homogeneous communication and computation costs, using the one-port model) in [BHR94], and the case of a star (heterogeneous communication and computation costs, one-port model) in [CR00]. In this section, we present their main results for bus and star architectures.

The notations used throughout this section are the following:

- $\alpha_i$ 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.

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. $T_i = T_f = \alpha_i w_i$, $\forall i$). 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,

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

if data is sent successively to $P_2, \ldots, P_n$. Closed forms can be obtained for both the $\alpha_i$’s and $T_f$. These closed form are rather complicated, although the method for obtaining them is elementary, and we refer the reader to [SRL98] 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 [OCL00]. 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 $\alpha_i$'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.

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

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

Bertsimas and Gamarnik [BG99] introduce a scheduling algorithm which is asymptotically optimal when \( n = \sum_{(k, l) \in P} n_{kl} \to +\infty \). 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 \( v_j \), \( \forall (i, j) \in E, \forall (k, l) \in P \). 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 timestep. We obtain the following relaxed linear program:

**Minimize** \( C_{\text{max}} \),

**subject to**

1. \( \sum_{i, (k, l) \in E} x_{ki}^{kl} = n_{kkl} \quad \forall (k, l) \in P \)
2. \( \sum_{i, (k, l) \in E} x_{ik}^{kl} = n_{ikl} \quad \forall (k, l) \in P \)
3. \( \sum_{j, (i, j) \in E} x_{ij}^{kl} = \sum_{r, (i, r) \in E} x_{ir}^{kl} \quad \forall (k, l) \in P, \forall i \neq k, l \)
4. \( C_{ij} \leq C_{\text{max}} \quad \forall (i, j) \in E \)
5. \( x_{ij}^{kl} \geq 0, \quad C_{ij} \geq 0 \quad \forall (k, l) \in P, \forall (i, j) \in E \)

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 \( C_{\text{max}} \). 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 \( \Omega \) be the length of a phase (to be determined later) and let

\[
e_{ij}^{kl} = \left\lfloor \frac{x_{ij}^{kl} \Omega}{C_{\text{max}}} \right\rfloor, \quad \forall (k, l) \in P, \forall (i, j) \in E,
\]
be the number of packets (rounded from below) of type \((k,l)\) which circulate on the edge 
\((i,j)\) during \(\Omega\) time-steps in the relaxed problem. The algorithm proposed in [BG99] is the following:

**Input** Compute the optimal value \(C_{\text{max}}\) from the relaxed linear program.

**Step 1** During each phase \([\Omega, (l+1)\Omega]\), where \(l = 0, \ldots, \lceil \frac{C_{\text{max}}}{\Omega} \rceil - 1\), 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 \(a_{ij}^l\).

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

It can be proven that at time-step

\[
\left( \frac{C_{\text{max}}}{\Omega} + 1 \right) \Omega + |E||V|\left( \frac{C_{\text{max}}}{\Omega} |P| + |P| + \Omega \right),
\]

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 \(\left( \frac{C_{\text{max}}}{\Omega} + 1 \right) \Omega\), the number of packets that have not reached their destination is bounded by \(|E|\left( \frac{C_{\text{max}}}{\Omega} |P| + |P| + \Omega \right)\). These packets are routed sequentially on a path of length at most \(|V|\), hence the duration of Step 2 is not larger than \(|E||V|\left( \frac{C_{\text{max}}}{\Omega} |P| + |P| + \Omega \right)\). If we choose \(\Omega\) of the order of \(\sqrt{C_{\text{max}}}\), the makespan of the schedule is \(C_{\text{max}} + O(\sqrt{C_{\text{max}}}\), 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 31):

- 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 being associated to a file of type \((k,l)\) to be sent from \(T_k\) to \(T_l\)).

- 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, $Sent(P_i, P_j, T_k, T_l)$ is the number of data files of this type sent along the edge $(P_i, P_j)$ per time-unit, with $s(P_i, P_j, T_k, T_l) = Sent(P_i, P_j, T_k, T_l) \times data_{k,l} \times c_{i,j}$.

- $\alpha(P_i, T_k)$ is the fraction of time spent each time-unit by $P_i$ to compute tasks of type $T_k$. Similarly, $Cons(P_i, T_k)$ is the number of tasks of this type consumed by $P_i$ each time-unit, with $\alpha(P_i, T_k) = Cons(P_i, T_k) \times w_i, k$.

- Finally, we add two fictitious tasks $T_{\text{begin}}$ and $T_{\text{end}}$ to the task graph. $T_{\text{begin}}$ is the predecessor of all input tasks in $G$ (tasks without any predecessor in $G$). The execution time of $T_{\text{begin}}$ by any processor is equal to 0, and the communication volume along any edge from $T_{\text{begin}}$ to an input task is also 0. Similarly, $T_{\text{end}}$ is the successor of all output tasks in $G$.

Let $P_{\text{ms}}$ 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:
\[
\text{Maximize } \sum_i \text{Cons}(P_i, T_{\text{end}}), \\
\text{subject to} \\
(1) \forall i, k, \quad 0 \leq \alpha(P_i, T_k) \leq 1 \\
(2) \forall i, j, k, l, \quad 0 \leq s(P_i, P_j, T_k, T_l) \leq 1 \\
(3) \forall i, j, k, l, \quad s(P_i, P_j, T_k, T_l) = \text{Sent}(P_i, P_j, T_k, T_l) \ast \text{data}_{k,l} \ast c_{i,j} \\
(4) \forall i, k, \quad \alpha(P_i, T_k) = \text{Cons}(P_i, T_k) \ast w_i, k \\
(5) \forall i, \sum_{P_j \in P_i} \sum_{(k,l) \in C} s(P_j, P_i, T_k, T_l) \leq 1 \\
(6) \forall i, \sum_{P_j \in P_i} \sum_{(k,l) \in C} s(P_j, P_i, T_k, T_l) \leq 1 \\
(7) \forall i, \sum_{T_k \in T} \alpha(P_i, T_k) \leq 1 \\
(8) \forall i, \text{Cons}(P_i, T_{\text{begin}}) = 0 \\
(9) \forall i, j, k, \quad s(P_i, P_j, T_k, T_{\text{end}}) = 0 \\
(10) \forall i, k, l, \quad \sum_{P_j \in P_i} \text{Sent}(P_j, P_i, T_k, T_l) + \text{Cons}(P_i, T_k) = \\
\sum_{P_j \in P_i} \text{Sent}(P_j, P_i, T_k, T_l) + \text{Cons}(P_i, T_k) \\
(11) \forall i, k \neq \text{begin}, l, \quad \sum_{P_j \in P_{\text{ms}}} \text{Sent}(P_j, P_{\text{ms}}, T_k, T_l) + \text{Cons}(P_{\text{ms}}, T_k) = \\
\sum_{P_j \in P_{\text{ms}}} (\text{Sent}(P_{\text{ms}}, P_j, T_k, T_l) + \text{Cons}(P_{\text{ms}}, T_l)) \\
\]

The objective function is equal to the number of copies of task \( T_{\text{end}} \) executed per time-step. Because of the dependences, the availability of a copy of \( T_{\text{end}} \) means that the whole task graph instance has been executed. Equations (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 [BBLR02, BLR02] 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 asymptotically 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 [Ber99]. 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 [AS07, CZL97]. 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 [WSH99]. 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 [FJJ+01, NZ01] or Effective Network View [SBW99]. 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 [Cas01], 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 [CLZ00]. An extension of SimGRID to decentralized schedulers and realistic platforms is currently under development [LL02].

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


