Scheduling strategies for master-slave tasking on heterogeneous processor grids

Cyril Banino, Olivier Beaumont, Arnaud Legrand, Yves Robert

Laboratoire de l'Informatique du Parallélisme

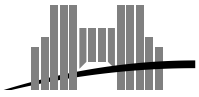
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Scheduling strategies for master-slave tasking on heterogeneous processor grids

Cyril Banino,
Olivier Beaumont,
Arnaud Legrand and
Yves Robert

March 2002

Research Report N° 2002-12



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

Scheduling strategies for master-slave tasking on heterogeneous processor grids

Cyril Banino,
Olivier Beaumont,
Arnaud Legrand and
Yves Robert

March 2002

Abstract

In this paper, we consider the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous "grid" computing platform. We use a non-oriented graph to model a grid, where resources can have different speeds of computation and communication, as well as different overlap capabilities. We show how to determine the optimal steady-state scheduling strategy for each processor (the fraction of time spent computing and the fraction of time spent communicating with each neighbor). This result holds for a quite general framework, allowing for cycles and multiple paths in the interconnection graph, and allowing for several masters.

Because spanning trees are easier to deal with in practice (there is a single path from the master to each node), a natural question arises: how to extract the best spanning tree, i.e. the one with optimal steady-state throughput, out of a general interconnection graph? We show that this problem is NP-hard. Even worse, we show that there exist heterogeneous networks for which the optimal spanning tree has a throughput which is arbitrarily bad in front of the throughput that can be achieved by the optimal (multiple-path) solution. Still, we introduce and compare several low-complexity heuristics to determine a sub-optimal spanning tree. Fortunately, we observe that the best heuristics do achieve an excellent performance in most experiments.

Keywords: heterogeneous processors, master-slave tasking, communication, spanning trees, complexity.

Résumé

Dans ce rapport, nous nous intéressons au problème de l'allocation d'un grand nombre de taches indépendantes et de taille identiques sur des plateformes de calcul hétérogènes comme la fameuse *computing grid*. Nous utilisons des graphes non-orientés pour modéliser les plateformes dont les ressources peuvent avoir des vitesses de calcul ou de communication différentes les unes des autres ainsi que diverses possibilités de recouvrement. Nous montrons comment déterminer le régime permanent optimal pour chaque processeur (c'est-à-dire la fraction de temps passée à calculer et celles passées à communiquer avec chacun de ses voisins). Ces résultats restent valables dans un cadre plus général où l'on autorise les cycles et les chemins multiples dans le graphe d'interconnexion, ainsi que l'existence de plusieurs maîtres.
Les arbres étant cependant plus facile à utiliser en pratique (il n'y a qu'un seul chemin du maître à un autre processeur), il est naturel de se demander comment extraire du réseau d'interconnexion le meilleur arbre couvrant, c'est-à-dire celui ayant le meilleur rendement en régime permanent. Nous démontrons que ce problème est NP-difficile et qu'il n'est pas approximable à un facteur constant près : il existe une famille de graphes d'interconnexion dont l'arbre couvrant optimal a un rendement arbitrairement éloigné de celui qu'une solution utilisant plusieurs chemins peut obtenir. Nous introduisons et comparons cependant un certain nombre d'heuristiques de faible complexité et déterminant un arbre couvrant sous-optimal. Les meilleures heuristiques donnent d'excellents résultats dans la plupart des expériences.

Mots-clés: Ressources hétérogènes, maître/esclaves, communications, arbre couvrant, complexité.

Scheduling strategies for master-slave tasking on heterogeneous processor grids

C. Banino[1], O. Beaumont[1], A. Legrand[2] and Y. Robert[2]

1: LaBRI, UMR CNRS 5800, Domaine Universitaire, 33405 Talence Cedex, France

2: LIP, UMR CNRS-INRIA 5668
École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France

March 2002

Corresponding author: Yves Robert
LIP, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
Phone: + 33 4 72 72 80 37, Fax: + 33 4 72 72 80 80
E-mail: Yves.Robert@ens-lyon.fr

1 Introduction

In this paper, we deal with the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous "grid" computing platform. We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of an undirected graph. Each node is a computing resource (a processor, or a cluster, or whatever) capable of computing and/or communicating with its neighbors at (possibly) different rates.

We assume that one specific node, referred to as the master, initially holds (or generates the data for) a large collection of independent, identical tasks to be allocated on the grid. The question for the master is to decide which tasks to execute itself, and how many tasks to forward to each of its neighbors. Due to heterogeneity, the neighbors may receive different amounts of work (maybe none for some of them). Each neighbor faces in turn the same dilemma: determine how many tasks to execute, and how many to delegate to other processors.

The underlying interconnection network may be very complex and, in particular, may include multiple paths and cycles (just as the Ethernet does). The master may well need to send tasks along multiple paths to properly feed a very fast but remote computing resource. The master-slave scheduling problem for a general interconnection graph is to determine a steady state scheduling policy for each processor, i.e. the fraction of time spent computing, and the fraction of time spent sending or receiving tasks along each communication link, so that the (averaged) overall number of tasks processed at each time-step is maximum. In this paper, we solve the master-slave scheduling problem for general graphs, using a linear programming formulation (which nicely encompasses the situation where there are several masters instead of a single one).

The master-slave scheduling problem is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [23], factoring large numbers [9], the Mersenne prime search [22], and those distributed computing problems organized by companies such as Entropia [10]. Several papers [25, 24, 14, 12, 31, 3, 2] have recently revisited the master-slave paradigm for processor clusters or grids, and we refer to Section 7 for comparison and discussion.

This paper is a follow-on of recent work by Beaumont et al. [2], who solve the master-slave scheduling problem for a tree-shaped heterogeneous platform. Given an oriented spanning tree rooted at the master, they aim at determining the optimal steady-state scheduling strategy. Interestingly, it turns out that this strategy is *bandwidth-centric*: if enough bandwidth is available to the node, then all children are kept busy; if bandwidth is limited, then tasks should be allocated only to children

which have sufficiently fast communication times, in order of fastest communication time. Counter-intuitively, the maximum throughput in the tree is achieved by delegating tasks to children as quickly as possible, and not by seeking their fastest processing.

In a practical context, there are several reasons that can motivate to only use tree-shaped platforms. First of all, the unique route from the master to every processor will ease the implementation. Second, the bandwidth-centric strategy mentioned above can be computed locally, and is therefore very robust in front of possible variations in processor speeds and in communication bandwidths. Overall, the situation is similar to the situation for broadcast algorithms: while optimal solutions may use multiple paths to route different fragments of the message, practical implementations, such as the MPI [29] *one-to-all* routine, typically rely on spanning trees.

Given a network topology (that may well include cycles and multiple paths), how to extract the "best" spanning tree, i.e. the spanning tree which allows for the maximum number of tasks to be processed by all the computing resources? Given a tree, the result by Beaumont et al [2] enables to compute the best scheduling strategy for that tree, but is of no help to find the tree. Because there may exist an exponential number of trees rooted at the master, we cannot simply compute the best scheduling strategy for each tree, and then select the best result.

Given a general interconnection graph, we show that the problem of extracting the optimal spanning tree is NP-complete. Even worse, we show that there exist heterogeneous networks for which the optimal spanning tree has a throughput which is arbitrarily bad in front of the throughput that can be achieved by the optimal (multiple-path) solution. Still, we introduce and compare several low-complexity heuristics to determine a sub-optimal spanning tree. Fortunately, we observe that the best heuristics do achieve an excellent performance in most experiments.

The rest of the paper is organized as follows. In Section 2 we introduce our base model of communication and computation, and we formally state the master-slave scheduling problem for a general interconnection graph. We provide the optimal solution to this problem, using a linear programming approach. In Section 3 we discuss various extensions, first with several masters, and then with different hypotheses on the overlapping capabilities. Section 4 is theoretically oriented and provides the negative complexity results for the search of an optimal spanning tree: (i) NP-completeness of the problem and (ii) inapproximability of a general graph by any spanning tree. Section 5 deals with the design of five low-cost (polynomial) heuristics to determine a sub-optimal spanning tree. These heuristics are experi-

mentally compared in Section 6. Fortunately, we are able to report that the best two heuristics achieve very good performance in most cases, despite the negative theoretical predictions. We briefly survey related work in Section 7. Finally, we give some remarks and conclusions in Section 8.

2 The master-slave scheduling problem

In this section, we formally state the optimization problem to be solved. We start with the architectural model, next we explain how to compute the steady state, and finally we state the master-slave scheduling problem as a linear programming problem to be solved in rational numbers (hence a polynomial complexity).

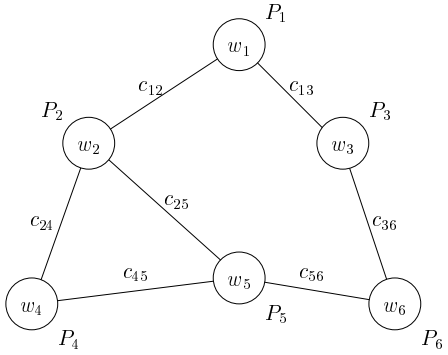


Figure 1: A graph labeled with node (computation) and edge (communication) weights.

2.1 Architectural model

The target architectural/application framework is represented by a node-weighted edge-weighted graph $G = (V, E, w, c)$, as illustrated in Figure 1. Let $p = |V|$ be the number of nodes. Each node $P_i \in V$ represents a computing resource of weight w_i, meaning that node P_i requires w_i units of time to process one task (so the smaller w_i, the faster the processor node P_i). There is a *master* processor, i.e. a node P_m which plays a particular role. P_m initially holds the data for a large (say infinite) collection of independent tasks to be executed. Tasks are atomic, their computation or communication cannot be preempted. A task represents the granularity of the application.

Each edge $e_{ij} : P_i \to P_j$ is labeled by a value c_{ij} which represents the time needed to communicate the data for one task between P_i and P_j, in either direction: we assume that the link between P_i and P_j is bidirectional and symmetric, i.e. that it takes the same amount of time to send (the data for) one task from P_i to P_j than in the reverse direction, from P_j to P_i. A variant would be to assume two unidirectional links, one in each direction, with possibly different label values, and we explain below how to modify the formulas to handle this variant. If there is no communication link between P_i and P_j we let $c_{ij} = +\infty$, so that $c_{ij} < +\infty$ means that P_i and P_j are neighbors in the communication graph. Note that we can include in c_{ij} the time needed for the receiving processor to return the result to the sending processor when it is finished. For the purpose of computing steady-state behavior, it does not matter what fraction of the communication time is spent sending a problem and what fraction is spent receiving the results. To simplify the exposition, we will henceforth assume that all the time is spent sending the task data, and no time is needed to communicate the results back. We assume that all w_i are positive rational numbers. We disallow $w_i = 0$ since it would permit node P_i to perform an infinite number of tasks, but we allow $w_i = +\infty$; then P_i has no computing power but can still forward tasks to other processors. Similarly, we assume that all c_{ij} are positive rational numbers (or equal to $+\infty$ if there is no link between P_i and P_j).

There are several scenarios for the operation of the processors, which are surveyed in Section 3. In this section, we concentrate on the *full overlap, single-port* model, where a processor node can simultaneously receive data from one of its neighbor, perform some (independent) computation, and send data to one of its neighbor. At any given time-step, there are at most two communications involving a given processor, one in emission and the other in reception.

We state the communication model more precisely: if P_i sends a task to P_j at time-step t, then

- P_j cannot start executing this task before time-step $t + c_{ij}$,
- P_j can neither initiate another receive operation nor start the execution of the task before time-step $t + c_{ij}$ (but it can perform a send operation and independent computation),
- P_i cannot initiate another send operation before time-step $t + c_{ij}$ (but it can perform a receive operation and independent computation).

See Figure 2 for an example, where P_1 is the master.

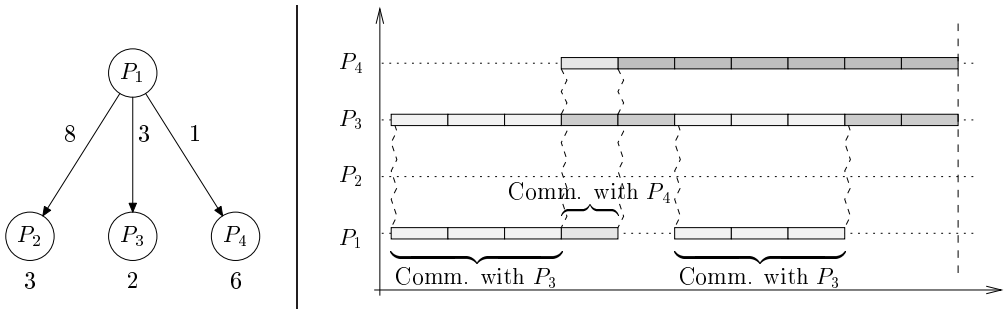


Figure 2: Example of execution: at most three tasks can be scheduled within $T = 10$ time-units.

2.2 Steady-state operation

Given the resources of a weighted graph G operating under the base model, we aim at determining the best steady-state scheduling policy. After a start-up phase, we want the resources to operate in a periodic mode. This makes very good sense if there is a large number of tasks to process, as typical applications would require: otherwise why bother dispatching them on the grid?

To formally define the steady-state, we need a couple of notations. Let $n(i)$ denote the index set of the neighbors of processor P_i. During one time unit:

- α_i is the fraction of time spent by P_i computing
- s_{ij} is the fraction of time spent by P_i sending tasks to each neighbor processor P_j, $j \in n(i)$, i.e. for each $e_{ij} \in E$
- r_{ij} is the fraction of time spent by P_i receiving tasks from each neighbor processor P_j, $j \in n(i)$, i.e. for each $e_{ij} \in E$

We search for rational values of all these variables. The first set of constraints is that they all must belong to the interval $[0, 1]$, as they correspond to the activity during one time unit:

$$\forall i, 0 \leq \alpha_i \leq 1 \tag{1}$$
$$\forall i, \forall j \in n(i), 0 \leq s_{ij} \leq 1 \tag{2}$$
$$\forall i, \forall j \in n(i), 0 \leq r_{ij} \leq 1 \tag{3}$$

The second set of constraints is that the number s_{ij}/c_{ij} of tasks sent by P_i to P_j is equal to the number of tasks r_{ji}/c_{ij} received by P_j from P_i:

$$\forall e_{ij} \in E, s_{ij} = r_{ji} \tag{4}$$

Remember that the communication graph is assumed to be symmetric: $e_{ij} \in E \Rightarrow e_{ji} \in E$, and therefore we also have $s_{ji} = r_{ij}$. It may well be the case that each link will only be used in one direction in the final solution, i.e. that either r_{ij} or s_{ij} will be zero, but we cannot guarantee this *a priori*.

There are specific constraints for the base model:

One-port model for outgoing communications Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation

$$\forall i, \sum_{j \in n(i)} s_{ij} \leq 1 \tag{5}$$

One-port model for incoming communications Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation

$$\forall i, \sum_{j \in n(i)} r_{ij} \leq 1 \tag{6}$$

Full overlap because of the full overlap hypothesis, there is no further constraint on α_i: $0 \leq \alpha_i \leq 1$, and $\alpha_i = 1$ would mean that P_i is kept processing tasks all the time.

Limited bandwidth This constraint is due to our hypothesis that the same link e_{ij} may be used in both directions simultaneously. We have to guaranteed that the link bandwidth is not exceeded. The constraint translates into:

$$\forall e_{ij} \in E, s_{ij} + r_{ij} \leq 1 \tag{7}$$

We can slightly refine the model, by introducing b_{ij}, the link bandwidth, expressed in tasks per second. Each time unit, there are $\frac{s_{ij}}{c_{ij}}$ tasks sent by P_i to P_j, and $\frac{r_{ij}}{c_{ij}}$ tasks received by P_i to P_j, so constraint (7) writes

$$\forall e_{ij} \in E, \frac{s_{ij}}{c_{ij}} + \frac{r_{ij}}{c_{ij}} \leq b_{ij}$$

This amounts to let $b_{ij} = 1/c_{ij}$ in equation (7).

The last constraints deals with *conservation laws*: for every processor P_i which is not the master, the number of tasks received by P_i, i.e. $\sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}}$, should be equal to the number of tasks that P_i consumes itself, i.e. $\frac{\alpha_i}{w_i}$, plus the number of tasks forwarded to its neighbors, i.e. $\sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}}$. We derive the equation:

$$\forall i \neq m, \sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}} \tag{8}$$

It is important to understand that Equation (8) really applies to the steady-state operation. We can assume an initialization phase, during which tasks are forwarded to processors, and no computation is performed. Then, during each time-period in steady-state, each processor can simultaneously perform some computations, and send/receive some other tasks. This is why Equation (8) is sufficient, we do not have to detail which operation is performed at which time-step, because they all commute.

Equation (8) does not hold for the master processor P_m, because it holds an infinite number of tasks. Without loss of generality, we can enforce that $r_{mj} = 0$ for all $j \in n(m)$: the master does not need to receive any task from its neighbors.

Note that it would be easy to handle unidirectional links: if $e_{ij} : P_i \to P_j$ is unidirectional (that is, if for some reason P_i can send tasks to P_j but not the other way round), we let $r_{ij} = s_{ji} = 0$ and we suppress equation (7), which is automatically fulfilled. Similarly, it is straightforward to replace each bidirectional link e_{ij} by two oriented arcs a_{ij} (from P_i o P_j) and a_{ji} (from P_j to P_i), respectively weighted with c_{ij} and c_{ji}.

The equations above constitute a linear programming problem, whose objective function is the number of tasks consumed within one unit of time, i.e. the throughput $n_{\text{task}}(G) = \sum_i \frac{\alpha_i}{w_i}$. Here is a summary:

MASTER SLAVE SCHEDULING PROBLEM MSSP(G)

Maximize

$$n_{\text{task}}(G) = \sum_{i=1}^{p} \frac{\alpha_i}{w_i},$$

subject to

$$\begin{cases} \forall i, & 0 \leq \alpha_i \leq 1 \\ \forall i, & \forall j \in n(i), 0 \leq s_{ij} \leq 1 \\ \forall i, & \forall j \in n(i), 0 \leq r_{ij} \leq 1 \\ \forall e_{ij} \in E, & s_{ij} = r_{ji} \\ \forall i, & \sum_{j \in n(i)} s_{ij} \leq 1 \\ \forall i, & \sum_{j \in n(i)} r_{ij} \leq 1 \\ \forall e_{ij} \in E, & s_{ij} + r_{ij} \leq 1 \\ \forall i \neq m, & \sum_{j \in n(i)} \frac{r_{ij}}{c_{ij}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{ij}}{c_{ij}} \\ \forall j \in n(m), & r_{mj} = 0 \end{cases}$$

Note that we can enforce $\alpha_m = 1$, because the master will keep on processing tasks all the time, but this condition will automatically be fulfilled by the solution.

We can state the first result of this paper:

Theorem 1. *The solution to the previous linear programming problem provides the optimal solution to* MSSP(G)

Because we have a linear programming problem in rational numbers, we obtain rational values for all variables in polynomial time (polynomial in $|V| + |E|$, the size of the heterogeneous platform). When we have the optimal solution, we take the least common multiple of the denominators, and thus we derive an integer period T for the steady-state operation.

Finally, we point out that we can restrict to solutions where each link is used only in one direction. Although there may exist optimal solutions of MSSP(G) for which it is not the case, we can always transform such solutions into solutions where each link is used only in one direction (without changing the throughput): if both r_{ij} and s_{ij} are non-zero, with, say, $r_{ij} \geq s_{ij}$, use $r'_{ij} = r_{ij} - s_{ij}$ and $s'_{ij} = 0$ to derive an equivalent solution.

2.3 Example

Consider the toy example of Figure 3, with $p = 4$ processors. If we feed the values w_i and c_i into the linear program, and compute the solution using a tool like the Maple

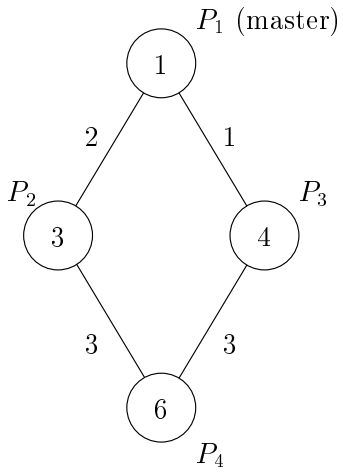


Figure 3: An example with four processors.

simplex package [7], we obtain the optimal throughput $n_{\text{task}}(G) = \frac{7}{4}$. This means that the whole platform is equivalent to a single processor with processing capability $w = \frac{1}{n_{\text{task}}(G)} = \frac{4}{7}$, i.e. capable of processing 7 tasks every 4 seconds.

With the values of α_i, s_{ij} and r_{ij} returned in the solution of the linear program, we retrieve the periodic steady-state behavior. Every 12 time-units:

- The master processor P_1 computes twelve tasks ($\alpha_1 = 1$), sends two tasks to P_2 (in 4 time-steps, $s_{12} = 1/3$), and sends seven tasks to P_3 (in 7 time-steps, $s_{13} = 7/12$)
- Processor P_3 receives seven tasks from the master P_1 (in 7 times-steps, $r_{31} = 7/12$), computes three tasks ($\alpha_3 = 1$) and sends four tasks to P_4 (in 12 time-steps, $s_{34} = 1$)
- Processor P_4 receives four tasks from P_3 (in 12 time-steps, $r_{43} = 1$), computes two tasks ($\alpha_4 = 1$) and sends two tasks to P_2 (in 6 time-steps, $s_{42} = 1/2$)
- Processor P_2 receives four tasks, two from the master P_1 (in 4 time-steps, $r_{21} = 1/3$) and two from P_4 (in 6 time-steps, $r_{24} = 1/2$), and it computes four tasks ($\alpha_2 = 1$).

This makes a total of $12 + 3 + 2 + 4 = 21$ tasks every 12 time-steps, and we do retrieve the value $n_{\text{task}}(G) = \frac{21}{12} = \frac{7}{4}$. This steady-state is illustrated in Figure 4.

Note that all processors are executing tasks all the time, so the solution achieves a full utilization of the computing resources. It is interesting to point out that P_2 receives its tasks along two paths, the first half directly from the master, and the second half being forwarded through P_3 and P_4.

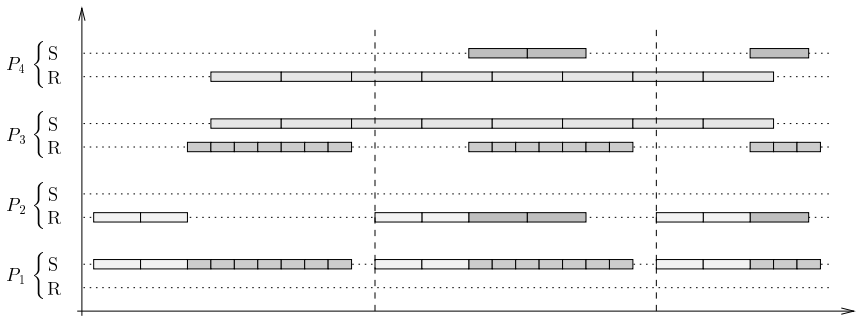


Figure 4: Steady-state for the example with four processors.

In the introduction, we briefly mentioned the difficulty of extracting a spanning tree of high throughput out of a given interconnection graph, and we come back to this point in Section 4. We can perform an exhaustive search for our little example, because there are only four possible trees rooted in P_1. To compute the throughput of a given tree, we can either use the linear programming approach, or traverse the tree using the bandwidth-centric algorithm of [2] (with a cost linear in the processor number). In the example, we obtain the following results:

1. If we suppress the edge between P_1 and P_2, the throughput of the corresponding tree T_1 is $n_{\text{task}}(T_1) = \frac{38}{24}$
2. If we suppress the edge between P_1 and P_3, the throughput of the corresponding tree T_2 is $n_{\text{task}}(T_2) = \frac{36}{24}$
3. If we suppress the edge between P_2 and P_4, the throughput of the corresponding tree T_3 is $n_{\text{task}}(T_3) = \frac{41}{24}$
4. If we suppress the edge between P_3 and P_4, the throughput of the corresponding tree T_4 is $n_{\text{task}}(T_4) = \frac{39}{24}$

We see that the third tree T_3 is the best one, with a throughput $n_{\text{task}}(T_3) = \frac{41}{24}$ very close to the optimal solution $n_{\text{task}}(G) = \frac{42}{24}$ for the whole graph.

3 Extensions

In this section we discuss several extensions: with several masters, and with other models of operation than the base model.

3.1 With several masters

The extension for several masters is straightforward. Assume that there are k masters P_{m_1}, P_{m_2}, ..., P_{m_k}, each holding (the initial data for) a large collection of tasks. For each index m_q, $1 \leq q \leq k$:

1. Suppress equation (8) for $i = m_q$ (the conservation law does not apply to a master)
2. Add the constraints $r_{m_q,j} = 0$ for all $j \in m(q)$ (a master does not need to receive any task)

We then solve the new MSSP(G) problem.

3.2 With other models

We rely on the classification proposed by Beaumont et al [2]:

$\mathcal{M}(r^*\|s^*\|w)$**: Full overlap, multiple-port** In this first model, a processor node can simultaneously receive data from all its neighbors, perform some (independent) computation, and send data to all of its neighbors. This model is not realistic if the number of neighbors is large.

$\mathcal{M}(r\|s\|w)$**: Full overlap, single-port** In this second model, a processor node can simultaneously receive data from one neighbor, perform some (independent) computation, and send data to one neighbor. At any given time-step, there are at most two communications taking place, one incoming and one outgoing. This model is representative of a large class of modern machines, and is the *base model* which we have already dealt with.

$\mathcal{M}(r\|s,w)$**: Receive-in-Parallel, single-port** In this third model, as in the next two, a processor node has one single level of parallelism: it can perform two actions simultaneously. In the $\mathcal{M}(s\|r,w)$ model, a processor can simultaneously receive data from one neighbor, and either perform some (independent) computation, or send data to one neighbor.

$\mathcal{M}(s\|r,w)$**: Send-in-Parallel, single-port** In this fourth model, a processor node can simultaneously send data to one neighbor and either perform some (independent) computation, or receive data from one neighbor.

$\mathcal{M}(w\|r,s)$**: Work-in-Parallel, single-port** In this fifth model, a processor node can simultaneously compute and execute a single communication, either sending to or receiving from one neighbor.

$\mathcal{M}(r,s,w)$**: No internal parallelism** In this sixth and last model, a processor node can only do one thing at a time: either receiving from one neighbor, or computing, or sending data to one neighbor. This is really the low-end computer!

3.2.1 Reduction for $\mathcal{M}(r^*\|s^*\|w)$, the Full overlap, multiple-port model

In this model, we allow for an unlimited number of simultaneous communications, either incoming or outgoing. It is quite easy to take this new constraint into account: simply suppress equations (5) and (6) in the linear program! Indeed, under the new model, equations (2) and (3) are sufficient to characterize the activity of each processor.

Instead of allowing an unlimited number of simultaneous communications, we could be more restrictive and restrict each processor to k_1 incoming and k_2 outgoing communications. In other words, there are k_1 receiving ports, and k_2 sending ports. Let r_{ij}^k be the time spent by processor P_i to receive tasks from processor P_j on receiving port k for $1 \leq k \leq k_1$. Similarly, let s_{ij}^k be the time spent by P_i to send tasks to P_j on sending port k, for $1 \leq k \leq k_2$, The new constraints simply are

$$\forall i, \forall k, 1 \leq k \leq k_1, 0 \leq r_{ij}^k \leq 1 \tag{9}$$

$$\forall i, \forall k, 1 \leq k \leq k_2, 0 \leq s_{ij}^k \leq 1 \tag{10}$$

3.2.2 Reduction for $\mathcal{M}(r\|s,w)$, the Receive-in-Parallel, single port model

This model is less powerful than the base model: processor can simultaneously receive a task from one of its neighbors, and either perform some computation, or send a task to one of its neighbors. To take this new constraint into account, simply replace equation (5) by

$$\forall i, \alpha_i + \sum_{j \in n(i)} s_{ij} \leq 1 \tag{11}$$

3.2.3 Reduction for $\mathcal{M}(s\|r, w)$, the Send-in-Parallel, single port model

In this model, a processor can simultaneously send a task to one of its neighbors, and either perform some computation, or receive a task from one of its neighbors. To take this new constraint into account, simply replace equation (6) by

$$\forall i, \alpha_i + \sum_{j \in n(i)} r_{ij} \leq 1 \tag{12}$$

3.2.4 Reduction for $\mathcal{M}(w\|r, s)$, the Work-in-Parallel, single port model

In this model, a processor can simultaneously perform some computation, and either receive from, or send a task to, one of its neighbors, To take this new constraint into account, simply replace equations (5) and (6) by

$$\forall i, \sum_{j \in n(i)} s_{ij} + \sum_{j \in n(i)} r_{ij} \leq 1 \tag{13}$$

3.2.5 Reduction for $\mathcal{M}(r, s, w)$, the No internal parallelism model

In this model, a processor can only do one thing at a time: receive, send or compute tasks. This time, we have to replace the three equations (1), (5) and (6) by

$$\forall i, \alpha_i + \sum_{j \in n(i)} s_{ij} + \alpha_i + \sum_{j \in n(i)} s_{ij} \leq 1 \tag{14}$$

3.2.6 Strongly heterogeneous platforms

Finally, it is important to point out that the processor nodes may operate under different modes. Instead of writing the same equations for each node, we pick up different equations for each node, those corresponding to the desired operation modes.

4 Spanning trees

For a general interconnection graph, the solution of the linear program may lead to the use of multiple paths (this is the case for the toy example of Section 2.3. As already mentioned, it may be of interest to extract the best spanning tree (the one with maximum throughput) out of the graph. Using a tree greatly simplifies the implementation (because of the unique route from the master to any processor). Also, the bandwidth-centric algorithm presented in [2] is local and demand-driven, therefore is very robust to small variations in resources capabilities.

This section provides "negative" results: first, extracting the best tree is NP-hard. But even if we are ready to pay a high (exponential) cost to determine the best tree, there exist graphs for which the throughput of the best tree is arbitrarily bad in front of the throughput that can be achieved while the whole graph. In practice however, low-costs heuristics can be derived to determine sub-optimal but efficient spanning trees: see Sections 5 and 6.

4.1 Finding the best spanning tree

Our aim is to find the spanning tree that maximizes the throughput, i.e. the number of tasks that can be processed within one unit of time at steady state. Formally, we can state the problem as follows.

Definition 1 (BEST-TREE(G)). *Let $G = (V, E, w, c)$ be the node-weighted edge-weighted graph representing the architectural framework. Find the tree $T = (V, E', w, c)$, sub-graph of G, rooted at the master, such that the number of tasks $n_{task}(T)$ that can be processed in steady-state within one time-unit, using only those edges of the tree, is maximized.*

The associated decision problem is the following:

Definition 2 (BEST-TREE-DEC(G,α)). *Let $G = (V, E, w, c)$ be the node-weighted edge-weighted graph representing the architectural framework. Is there a tree $T = (V, E', w, c)$, sub-graph of G, rooted at the master, such that $n_{task}(T) \geq \alpha$?*

Theorem 2. BEST-TREE-DEC(G,α) *is NP-complete.*

Proof. The problem BEST-TREE-DEC(G,α) obviously belongs to the class NP. We prove its NP-completeness by reduction to 2-PARTITION, which is known to be NP-Complete [11]. Consider the following instance of 2-PARTITION: given

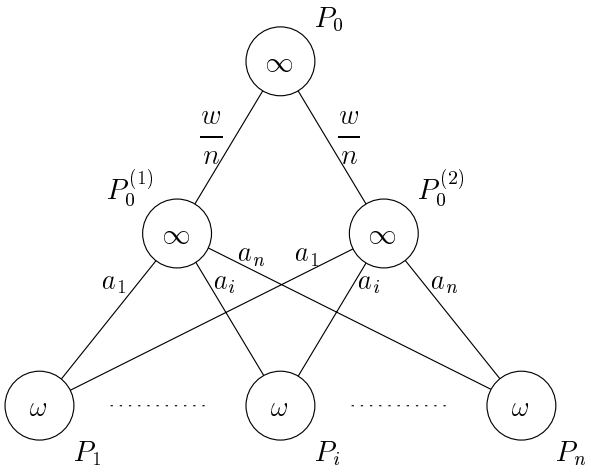


Figure 5: Instance of BEST-TREE-DEC used for the reduction.

$a_1, a_2, \ldots, a_n$ n positive integers, is there a partition of $[1, n]$ into two subsets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \ ?$$

We build the the following instance of BEST-TREE-DEC(G, α). Let G be the graph depicted in Figure 5. The master node P_0 has two neighbors, $P_0^{(1)}$ and $P_0^{(2)}$. The weight of the two edges from P_0 to its neighbor is 0. $P_0^{(1)}$ and $P_0^{(2)}$ have the same n neighbors P_i, $1 \leq i \leq n$, which are the only nodes with processing power. We let $w_i = w = \frac{1}{2}\sum_{i=1}^{n} a_i$ for $1 \leq i \leq n$. The weight of the edge between $P_0^{(1)}$ and P_i is the same as the weight of the edge between $P_0^{(2)}$ and P_i, and is equal to a_i, for $1 \leq i \leq n$. Finally, we let $\alpha = \frac{n}{w}$. Clearly, the size of this instance is polynomial (even linear) in the size of the original instance of 2-PARTITION.

We prove that there exists a solution to the original instance of 2-PARTITION if and only if there exists a solution to the instance of BEST-TREE-DEC(G, α) described above.

Consider a solution of BEST-TREE-DEC such that $n_{task}(T) \geq \alpha$. Any tree, sub-graph of G, and rooted at the master, can be uniquely described by a partition of the leaves P_i, $1 \leq i \leq n$, according to their father. Let us denote, for $i \in \{1, 2\}$, by $\mathcal{A}_i$ the set of indices of the leaves whose father is $P_0^{(i)}$, as depicted in Figure 6.

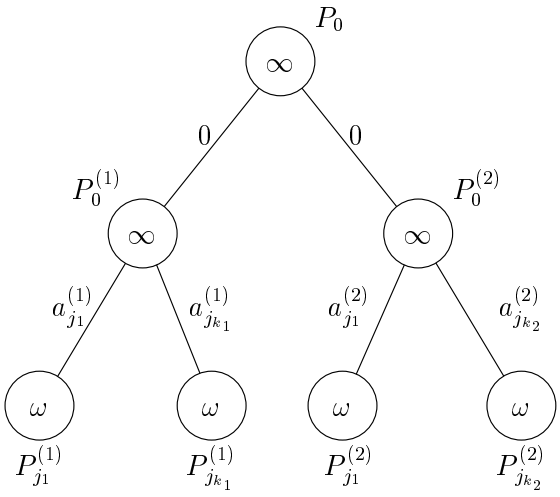


Figure 6: Description of a tree T extracted from G.

In order to compute $n_{task}(T)$, we refer the reader to [2], where formulae are given to compute the optimal throughput for a general tree. Let $j_1^{(i)} \leq \ldots \leq j_{k_i}^{(i)}$ be the set of indices of $\mathcal{A}_i$. Let $k_i^{'}$ and $k_i^{''}$, if they exist, be the indices of $\mathcal{A}_i$ such hat

$$\sum_{k=1}^{k_i^{'}} \frac{a_{j_k^{(i)}}}{w} \leq 1 \text{ and } \sum_{k=1}^{k_i^{''}} \frac{a_{j_k^{(i)}}}{w} > 1.$$

Then, if T_i denotes the subtree of T rooted in $P_0^{(i)}$, we have for $i \in \{1, 2\}$:

- If $k_i^{'} = k_i$, then $n_{task}(T_i) = \frac{k_i}{w}$
- If $k_i^{'} < k_i$, then $n_{task}(T_i) = \frac{k_i^{'}}{w} + \frac{1 - \sum_{k=1}^{k_i^{'}} \frac{a_{j_k^{(i)}}}{w}}{a_{k_i^{''}}}$ and, since $\sum_{k=1}^{k_i^{'}} \frac{a_{j_k^{(i)}}}{w} + \frac{a_{k_i^{''}}}{w} > 1$, then
$$\frac{1 - \sum_{k=1}^{k_i^{'}} \frac{a_{j_k^{(i)}}}{w}}{a_{k_i^{''}}} < \frac{1}{w} \text{ and } n_{task}(T_i) < \frac{k_i}{w}.$$

Thus $n_{task}(T_i) \leq \frac{k_i}{w}$ and $n_{task}(T_i) = \frac{k_i}{w}$ if and only if $\sum_{k=1}^{k_i} \frac{a_{j_k}^{(i)}}{w} \leq 1$.

Therefore, since $n_{task}(T) = n_{task}(T_1) + n_{task}(T_2)$, $n_{task}(T) \leq \frac{k_1}{w} + \frac{k_2}{w} = \frac{n}{w}$ and

$$n_{task}(T) = \frac{n}{w} \text{ iff } \forall i \in \{1,2\}, \quad \sum_{j \in \mathcal{A}_i} a_j \leq w.$$

Moreover, since $\sum_{j \in \mathcal{A}_1} a_j + \sum_{j \in \mathcal{A}_2} a_j = \sum_{j=1}^{n} a_j = 2w$, then

$$n_{task}(T) = \frac{n}{w} \text{ if and only if } \sum_{j \in \mathcal{A}_1} a_j = \sum_{j \in \mathcal{A}_2} a_j = w.$$

We have therefore proved that if there exists a solution to our instance of BEST-TREE-DEC, then there exists a solution to the original instance of 2-PARTITION.

Reciprocally, if there exists a solution to the original instance of 2-PARTITION, then we can check that the tree depicted in Figure 6 is able to process exactly $\frac{n}{w}$ tasks within a time unit.

This terminates the proof of the NP-completeness of BEST-TREE-DEC. ■

4.2 Inapproximability of a graph by a tree

One natural and interesting question is the following: how bad may the approximation of a graph by a tree be? The following theorem states the inapproximability of a general graph by a tree, with respect to throughput:

Theorem 3. *Given any positive integer K, there exists a graph G such that for any tree T, sub-graph of G and rooted at the master, we have*

$$\frac{n_{task}(G)}{n_{task}(T)} \geq K.$$

Proof. Consider the graph depicted in Figure 7.

One can easily check that, using all the communication resources, it is possible to process one task within each time unit, i.e. $n_{task}(G) = 1$. However, any tree T extracted from G is equivalent to the chain depicted in Figure 8, since $P_0^{'}$ is the only computing resource. Moreover, because of the slow link between P_i and $P_i^{'}$, the number of tasks that can be processed within one unit of time is bounded by $\frac{1}{K}$ and thus

$$\forall T, \quad \frac{n_{task}(G)}{n_{task}(T)} \geq K.$$

■

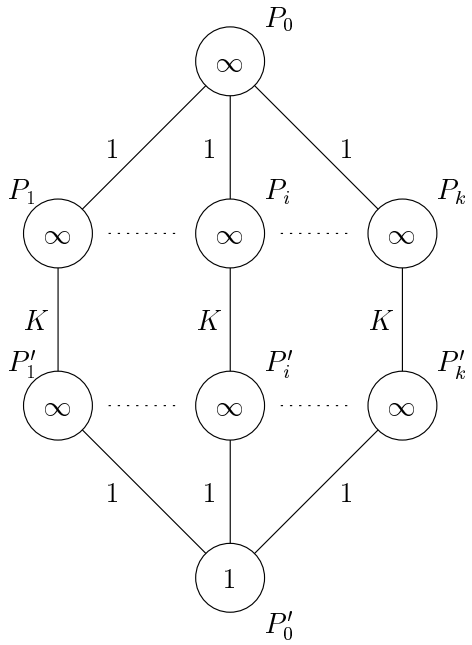


Figure 7: Graph G used for the proof of the inapproximability.

We have proved that the ratio between the number of tasks that can be processed using the graph and the number of tasks that can be processed using any extracted tree can be arbitrary large. Nevertheless, we show in Sections 5 and 6 that despite both the NP-completeness of the search of the best spanning tree, and the inapproximability of a graph by a spanning tree, it is possible to derive very efficient heuristics in practice.

5 Heuristics

In this section, we present several heuristics to extract a spanning tree with the highest possible throughput out of a general interconnection graph $G = (V, E, w, c)$. Given a spanning tree, we do not need the linear programming approach to compute its throughput: instead, we traverse the tree using the bandwidth-centric algorithm of [2], with a cost linear in $|V| + |E|$.

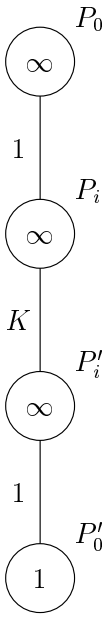


Figure 8: Description of a tree T extracted from G.

5.1 Greedy heuristics

Several greedy heuristics come to mind. We have selected (and implemented) the following:

Naive MST Given G, we compute the minimum spanning tree [8]. The edge weights are the communication parameters c_{ij}, and we do not use the computation parameters w_i at all. Given the non-oriented tree, we root it at the master and orient the edges accordingly.

Compute Tree We start from the master and take all the edges connecting to its neighbors. We sort these neighbors by non-decreasing w_i (faster processors first). For each neighbor P_i in sorted order, we consider its own neighbors and add the corresponding edge if that neighbor does not already belong to the tree. The process goes on until all nodes are included. This is a breadth-first traversal to grow the tree, where the more powerful neighbors of the current node are processed first.

C-to-C Tree This heuristic grows the tree similarly as the previous one, except that the sorted order is by non-decreasing values of the communication-to-computation ratios c_{ij}/w_j.

BW-centric Tree This heuristic is a variant of the compute tree heuristic: if node P_i is the current node, its neighbors P_j are still added according to the order of non-decreasing w_j, but only while the bandwidth-centric condition $\sum_j \frac{c_{ij}}{w_j} \leq 1$ holds. The idea is that the last neighbors will not be added to the tree if the bandwidth is saturated. At the end of the procedure there may remain isolated nodes, which we then connect to their closest neighbor (smallest value of the edge weight).

Again, once a tree is constructed, we compute its steady-state and its throughput using the algorithm in [2].

5.2 Heuristic based on the linear program

Our last heuristic is more costly, because it requires to solve the linear program for the initial interconnection graph. Once we have the solution, we weight each edge e_{ij} by the value $\frac{s_{ij}}{c_{ij}}$, which represents the average number of tasks which transit on the edge each second (in fact, because edges are bidirectional, we use $\frac{|s_{ij}-s_{ji}|}{c_{ij}}$). Given these weights, we extract a minimum spanning tree, which we call the LP Tree, and we compute its throughput as before.

6 Experiments

We have developed a software simulator that executes the heuristic algorithms of Section 5 and calculates the throughput for each of them. The inputs of the simulator are the number of nodes in the graph, the minimum degree, the maximum degree, the median degree, a probability function for the communication cost, and a probability function for the computation cost.

A random connected graph based on these parameters is generated. The simulator must also be given a list of communication-to-computation ratios to apply to computation costs. For each ratio and for each graph, the throughput of each heuristic is computed, and is compared to the optimal throughput that can be reached using the whole graph.

The following results are averaged values on 50 random graphs whose minimum degree is equal to 3, maximum degree is equal to 5 and average degree is equal to 4. The number of vertices range from 5 to 15.

In the following three simulations, the probability function for the communication costs follows a uniform distribution on the interval $[25, 35]$, and we let the probability function for the computation costs vary as

- a uniform distribution on the interval $[2.5, 3.5]$ in Figure 9(a). Communications are expensive, since the average communication-to-computation ratio is equal to 10.
- a uniform distribution on the interval $[25, 35]$ in Figure 9(b). Communication and computation costs follow the same distribution: the average communication-to-computation ratio is equal to 1.
- a uniform distribution on the interval $[250, 350]$ in Figure 9(c). Communications are cheap, since the average communication-to-computation ratio is equal to 0.1.

In the three figures, we depict the average ratio between the throughput of each heuristic and the optimal throughput of the whole graph.

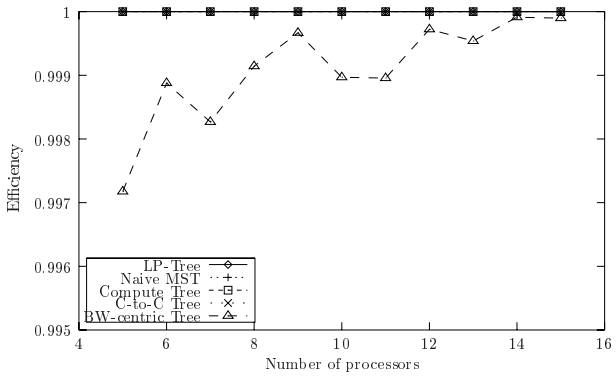
On each figure, LP-Tree and Naive MST are the most efficient heuristic and lead to trees whose throughput is very close to the optimal throughput of the whole graph.

7 Related problems

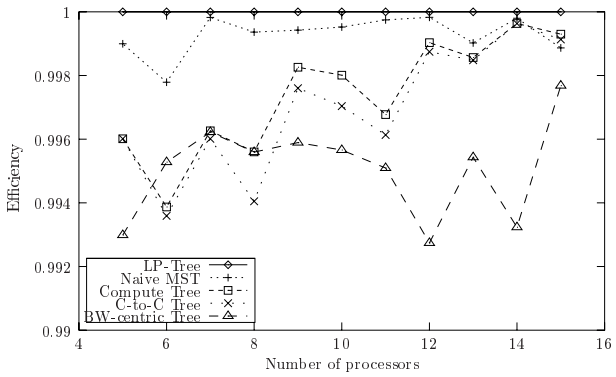
We classify several related papers along the following three main lines:

Scheduling task graphs on heterogeneous platforms Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors, see [20, 21, 30, 26, 6] among others. Unfortunately, all these heuristics assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications. Recent papers [15, 16, 28, 27] suggest to take communication contention into account. Among these extensions, scheduling heuristics under the one-port model [17, 18] are considered in [1]: just as in this paper, each processor can communicate with at most another processor at a given time-step.

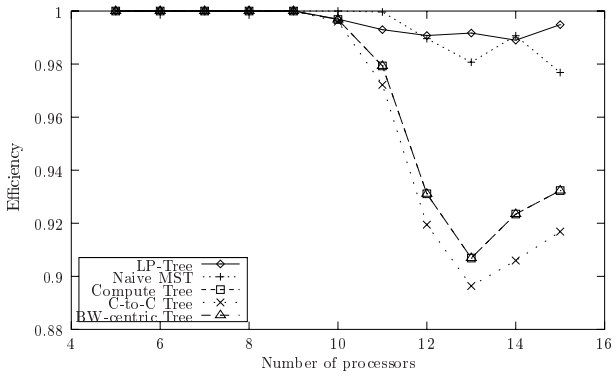
Collective communications on heterogeneous platforms Several papers deal with the complexity of collective communications on heterogeneous platforms:



(a) Computation costs range from 2.5 to 3.5 and communication costs range from 25 to 35. Since communications are very expensive, computations are not widely spread away from the master, and most optimal communication trees are a reduced depth.



(b) Both computation and communication costs range from 25 to 35. LP-Tree always leads to an optimal tree, and Naive MST behaves surprisingly well.



(c) Computation costs range from 250 to 350 and communication costs range from 25 to 35. Since communication are very cheap, spanning trees are very deep, and they are suboptimal compared to the throughput of the whole graph. Once more, Naive MST behaves surprisingly well.

Figure 9: Efficiency of the five heuristics : LP-Tree, Naive MST, Compute Tree, C-to-C Tree and BW-centric Tree.

broadcast and multicast operations are addressed in [5, 19], gather operations are studied in [13].

Master-slave on the computational grid Master-slave scheduling on the grid can be based on a network-flow approach [25, 24] or on an adaptive strategy [14]. Note that the network-flow approach of [25, 24] is possible only when using a full multiple-port model, where the number of simultaneous communications for a given node is not bounded. Enabling frameworks to facilitate the implementation of master-slave tasking are described in [12, 31].

Finally, from a theoretical point of view, it could be interesting to try to solve the complete scheduling problem associated to a general interconnection graph: instead of optimizing the steady-state throughput, how to maximize the total number of tasks processed within T time-units, for any time-bound T? Partial results are available in [3, 4], but the general problem looks quite challenging.

8 Conclusion

In this paper, we have dealt with master-slave tasking on a heterogeneous platform. We have shown how to determine the best steady-state scheduling strategy for a general interconnection graph, using a linear programming approach.

On one hand, we have derived negative theoretical results, namely that general interconnection graphs may be arbitrarily more powerful than spanning trees, and that determining the best spanning tree is NP-hard.

On the other hand, we have proposed several low-costs heuristics that achieve very good performances on a wide range of simulations. These positive experiments show that in practice, it is safe to rely on spanning trees to implement master-slave tasking.

This work can be extended in the following two directions:

- On the theoretical side, we could try to solve the problem of maximizing the number of tasks that can be executed within T time-steps, where T is a given time-bound. This scheduling problem is more complicated than the search for the best steady-state. Taking the initialization phase into account renders the problem quite challenging.

- On the practical side, we need to run actual experiments rather than simulations. Indeed, it would be interesting to capture actual architecture and application parameters, and to compare heuristics on a real-life problem.

References

[1] O. Beaumont, V. Boudet, and Y. Robert. 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.

[2] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2002*. IEEE Computer Society Press, 2002. Extended version available as LIP Research Report 2001-25.

[3] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In D.S. Katz, T. Sterling, M. Baker, L. Bergman, M. Paprzycki, and R. Buyya, editors, *Cluster'2001*, pages 419–426. IEEE Computer Society Press, 2001. Extended version available as LIP Research Report 2001-13.

[4] O. Beaumont, A. Legrand, and Y. Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. Technical Report 2002-07, LIP, ENS Lyon, France, February 2002.

[5] P.B. Bhat, V.K. Prasanna, and C.S. Raghavendra. Efficient collective communication in distributed heterogeneous systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*. IEEE Computer Society Press, 1999.

[6] Vincent Boudet and Yves Robert. Scheduling heuristics for heterogeneous processors. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 2109–2115. CSREA Press, 2001. Extended version available (on the Web) as Technical Report 2001-22, LIP, ENS Lyon.

[7] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *Maple Reference Manual*, 1988.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[9] James Cowie, Bruce Dodson, R.-Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Joerg Zayer. A world wide number field sieve factoring record: on to 512 bits. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - Asiacrypt '96*, volume 1163 of *LNCS*, pages 382–394. Springer Verlag, 1996.

[10] Entropia. URL: `http://www.entropia.com`.

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

[12] J.P Goux, S.Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society Press, 2000.

[13] J.-I. Hatta and S. Shibusawa. Scheduling algorithms for efficient gather operations in distributed heterogeneous systems. In *2000 International Conference on Parallel Processing (ICPP'2000)*. IEEE Computer Society Press, 2000.

[14] E. Heymann, M.A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

[15] L. Hollermann, T.S. Hsu, D.R. Lopez, and K. Vertanen. Scheduling problems in a practial allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[16] T.S. Hsu, J. C. Lee, D.R. Lopez, and W.A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[17] S.L. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, 1989.

[18] D.W. Krumme, G. Cybenko, and K.N. Venkataraman. Gossiping in minimal time. *SIAM J. Computing*, 21:111–139, 1992.

[19] R. Libeskind-Hadas, J.R.K. Hartline, P. Boothe, G. Rae, and J. Swisher. On multicast algorithms for heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 61(11):1665–1679, 2001.

[20] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[21] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, volume 1123 of *LNCS*, Lyon, France, August 1996. Springer Verlag.

[22] Prime. URL: `http://www.mersenne.org`.

[23] SETI. URL: `http://setiathome.ssl.berkeley.edu`.

[24] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.

[25] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.

[26] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[27] O. Sinnen and L. Sousa. 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)*, pages 382–387. IEEE Computer Society Press, 2001.

[28] O. Sinnen and L. Sousa. 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*, pages 166–170. Springer-Verlag LNCS 2150, 2001.

[29] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. The MIT Press, 1996.

[30] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1999.

[31] J.B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.