Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms

Cyril Banino, Olivier Beaumont, Larry Carter, *Fellow*, *IEEE*, Jeanne Ferrante, *Senior Member*, *IEEE*, Arnaud Legrand, and Yves Robert, *Member*, *IEEE*

Abstract—In this paper, we consider the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous computing platform. We use a nonoriented graph to model the platform, where resources can have different speeds of computation and communication. Because the number of tasks is large, we focus on the question of determining 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). In contrast to minimizing the total execution time, which is NP-hard in most formulations, we show that finding the optimal steady state can be solved using a linear programming approach and, thus, in polynomial time. Our result holds for a quite general framework, allowing for cycles and multiple paths in the interconnection graph, and allowing for several masters. We also consider the simpler case where the platform is a tree. While this case can also be solved via linear programming, we show how to derive a closed-form formula to compute the optimal steady state, which gives rise to a bandwidth-centric scheduling strategy. The advantage of this approach is that it can directly support autonomous task scheduling based only on information local to each node; no global information is needed. Finally, we provide a theoretical comparison of the computing power of tree-based versus arbitrary platforms.

Index Terms—Scheduling, heterogeneous, divisible load, steady-state, bandwidth-centric.

1 Introduction

In this paper, we deal with the master-slave paradigm on a heterogeneous platform, where resources have different speeds of computation and communication. More precisely, we tackle the problem of allocating a large number of independent, equal-sized tasks to a heterogeneous computing platform. We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of an undirected *platform 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. Think of a task as being associated with a file that contains all the data required for the execution of the task. The question for the master is to decide which tasks to execute itself, and how many tasks (i.e., task files) 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.

- C. Banino and O. Beaumont are with LaBRI, UMR CNRS 5800, Domaine Universitaire, 33405 Talence Cedex, France. E-mail: {Cyril.Banino, Olivier.Beaumont}@labri.fr.
- L. Carter and J. Ferrante are with the Computer Science Department, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114. E-mail: {carter, ferrante}@cs.ucsd.edu.
- A. Legrand and Y. Robert are with LIP, UMR CNRS-ENS Lyon-INRIA 5668, École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France. E-mail: {Arnaud.Legrand, Yves.Robert}@ens-lyon.fr.

Manuscript received 17 Feb. 2003; revised 7 July 2003; accepted 18 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0009-0203.

The master-slave scheduling problem is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [16] and those distributed computing projects organized by companies such as Entropia [8]. Several papers have recently revisited the master-slave paradigm for processor clusters or grids, and we refer to Section 6 for comparison and discussion.

Because the number of tasks to be executed on the computing platform is expected to be very large (otherwise, why deploy the corresponding application on computational grids?), we target *steady state* optimization problems rather than standard *makespan* minimization problems. While minimizing the makespan, i.e., the total execution time, is a NP-hard problem in most practical situations [9], [18], [7], it turns out that the optimal steady state can be characterized very efficiently, with low-degree polynomial complexity. The optimal steady state is defined as follows (see Section 2.2): For each processor, determine 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.

The main contribution of this paper is the determination of the optimal steady state in a wide general framework. We derive three main results that we summarize below:

- The first result holds for arbitrary platform graphs whose underlying interconnection network may be very complex and, in particular, may include multiple paths and cycles (just as the Internet does). The master may well need to send tasks along multiple paths to properly feed a very fast but remote computing resource. In this context, we compute the optimal steady state using a linear programming formulation, which nicely encompasses the situation where there are several masters instead of a single

one. We also show (see Section 5.1) that steady state scheduling is asymptotically optimal, among all possible schedules (not only periodic ones).

- The second result holds for a tree-shaped platform graph, i.e., when the underlying interconnection network is an oriented tree rooted at the master. For such platforms, we derive a closed-form formula that characterizes the optimal steady state, which can then be computed via a simple bottom-up traversal of the tree. Interestingly, the optimal steady state is achieved through a *bandwidth-centric* scheduling strategy: 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. The advantage of the bandwidth-centric approach is that it can directly support autonomous task scheduling based only on information local to each node; no global information is needed.
- The third result provides a comparison of the computing power of tree-shaped platforms versus arbitrary platform graphs. Given a network topology that may well include cycles and multiple paths, how does one extract the "best" spanning tree, i.e., the spanning tree rooted at the master which allows for the maximum number of tasks to be processed by all the computing resources? 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 compared with the throughput that can be achieved by the optimal (multiple-path) solution.

The rest of the paper is organized as follows: Section 2 is devoted to arbitrary platform graphs. We introduce the base model of communication and computation in Section 2.1. We formally state the equations governing the steady state in Section 2.2. In Section 2.3, we show how to determine the optimal steady state for an arbitrary platform graph, using a linear programming approach. Next, in Section 3, we deal with tree-shaped platform graphs: We provide a closed-form formula and a constructive method to compute the optimal steady state for such a platform. In Section 4, we discuss different hypotheses on the processing and communication capabilities of the computing resources: We show how to handle the case of processors operating with different characteristics and, in particular, different capabilities of overlapping communications with (independent) communications. In Section 5.1, we prove that steady state scheduling is always asymptotically optimal. Section 5.2 provides the negative complexity results stated above for the problem of extracting the optimal spanning tree (that of maximum steady state throughput) out of a given general platform. We briefly survey related work in Section 6. Finally, we give some remarks and conclusions in Section 7.

2 Optimal Steady State for General Platform Graphs

In this section, we formally state the optimization problem to be solved. We start with the architectural model in Section 2.1. Next, we explain all the equations governing the steady state operation on an arbitrary platform graph (Section 2.2). Then, in Section 2.3, we show how to cast the problem of determining the optimal steady state in terms of a linear programming problem to be solved in rational numbers (hence, of polynomial complexity). We deal with the extension to several masters in Section 2.3.3. We conclude this section by working out a little example in Section 2.4.

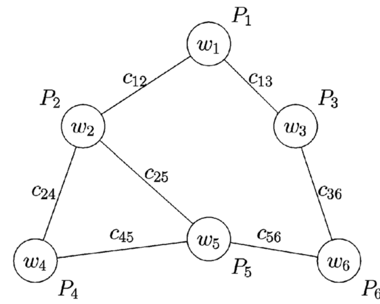

Fig. 1. A graph labeled with node (computation) and edge (communication) weights.

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 Fig. 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 unlimited) collection of independent tasks to be executed. Think of each task as being associated with a file that contains all the data required for the execution of the task. Tasks are atomic; their computation or communication cannot be preempted.

Each edge $e_{ij} : P_i \rightarrow P_j$ is labeled by a value c_{ij} which represents the time needed to communicate one (data file associated with a) 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 as 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 4. In this section, we concentrate on the *full overlap, single-port model*, which we also call the *base model*. In this model, a processor node can simultaneously receive data from one of its neighbors, perform some (independent) computation, and send data to one of its neighbors. At any given time-step, there are at most two communications involving a given processor, one sent and the other received.

We state the communication model more precisely: If P_i sends (the data file associated with) a task to P_j at time-step t, then: 1) P_j cannot start executing or sending this task before time-step $t + c_{ij}$; 2) P_j cannot initiate another receive operation before time-step $t + c_{ij}$ (but, it can perform a send operation and independent computation); and 3) P_i cannot initiate another send operation before time-step $t + c_{ij}$ (but, it can perform a receive operation and independent computation).

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.

To formally define the steady state, we need some notation. 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, for $j \in n(i)$, 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, for $j \in n(i)$, i.e., for each $e_{ij} \in E$.

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

$$\forall i, 0 \leq \alpha_i \leq 1, \quad (1)$$

$$\forall i, \forall j \in n(i), 0 \leq s_{ij} \leq 1, \quad (2)$$

$$\forall i, \forall j \in n(i), 0 \leq r_{ij} \leq 1. \quad (3)$$

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:

$$\forall e_{ij} \in E, s_{ij} = r_{ji}. \quad (4)$$

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

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

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

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

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 busy 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 guarantee that the link bandwidth is not exceeded. The constraint translates into:

$$\forall e_{ij} \in E, s_{ij} + r_{ij} \leq 1. \quad (7)$$

We can slightly reformulate (7) by introducing b_{ij}, the link bandwidth, expressed in tasks per second (i.e., $b_{ij} = 1/c_{ij}$). In 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) can be written

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

Conservation laws. The last constraints deal 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:

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

It is important to understand that (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 (8) is sufficient; we do not have to detail which operation is performed at which time-step because the tasks all commute, they are mutually independent.

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 tasks from its neighbors. Note that it would be easy to handle unidirectional links: If $e_{ij} : P_i \rightarrow P_j$ is unidirectional (that is, if for some reason P_i can send tasks to P_j, but not vice versa), we let $r_{ij} = s_{ji} = 0$ and we suppress (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 to P_j) and a_{ji} (from P_j to P_i), respectively, weighted with c_{ij} and c_{ji}.

2.3 Solution

2.3.1 Optimal Throughput

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 MSSG(G)

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

subject to

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

Note that we can enforce $\alpha_m = 1$ because the master will keep on processing tasks all the time, but we do not have to because this condition will automatically be fulfilled by the solution. We can state the first main result of this paper.

Theorem 1. *The solution to the previous linear programming problem provides the optimal solution to* MSSG(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 MSSG(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 nonzero, with, say, $r_{ij} \geq s_{ij}$, use $r'_{ij} = r_{ij} - s_{ij}$ and $s'_{ij} = 0$ to derive an equivalent solution.

2.3.2 Reconstructing the Schedule

There are several subtle points when reconstructing the actual periodic schedule. Once the linear programming problem is solved, we get the period T of the schedule, and the integer number of messages going through each link. First, because it arises from the linear program, $\log T$ is indeed a number polynomial in the problem size, but T itself is not. Hence, describing what happens at every time-step during the period would be exponential in the problem size, and we need a more "compact" description of the schedule. Second, we need to exhibit an orchestration of the message transfers where only independent communications, i.e., involving disjoint pairs of senders and receivers, can take place simultaneously.

Both problems are solved as follows: From our platform graph G, and using the result of the linear program, we build a bipartite graph: For each node P_i in G, we create two nodes P_i^{send} and P_i^{recv}. For each communication from P_i to P_j, we insert an edge between P_i^{send} and P_j^{recv}, which is weighted by the length of the communication. We are looking for a decomposition of this bipartite graph into a set of subgraphs where a node (sender or receiver) is occupied by at most one communication task. This means that at most one edge reaches each node in the subgraph. In other words, only communications corresponding to a matching in the bipartite graph can be performed simultaneously, and the desired decomposition of the graph is in fact an edge coloring. The weighted edge coloring algorithm of [15, vol. A, chapter 20] provides in time $\mathcal{O}(|E|^2)$ a polynomial number of matchings, which are used to perform the different communications, and which provides the desired polynomial-size description of the schedule. See [5] for further details. We point out that reconstructing the final schedule is much easier for tree-shaped platforms, as shown in Section 3.

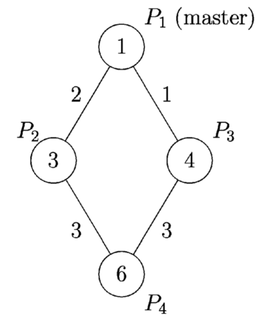

Fig. 2. An example with four processors.

Notice that it is necessary for each node to be able to hold in a buffer some number of tasks that it has received, but not yet computed or sent. Also, depending on the schedule, it may be necessary to preload some number of tasks into the node during the initialization period. The maximum number of such buffered tasks is at most T per node. The issue of limiting the number of buffered tasks to a more practical number is explored in [13].

2.3.3 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 (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 MSSG(G) problem.

2.4 Example

Consider the toy example of Fig. 2, with $p = 4$ processors (P_1 is the unique master). If we feed the values w_i and c_i into the linear program, and compute the solution using a tool such as the Maple simplex package, 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 seven tasks every four 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:

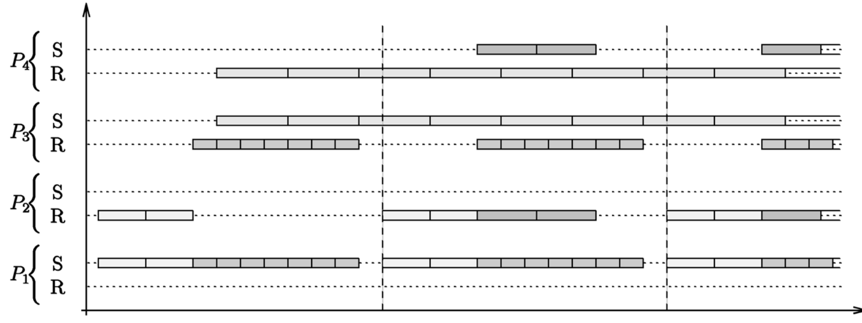

Fig. 3. Steady state for the example with four processors.

- The master processor P_1 computes 12 tasks ($\alpha_1 = 1$), sends two tasks to P_2 (in six time-steps, $s_{12} = 1/3$), and sends seven tasks to P_3 (in seven time-steps, $s_{13} = 7/12$).
- Processor P_3 receives seven tasks from the master P_1 (in seven 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 six 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 six 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 Fig. 3. 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.

In the introduction, we briefly mentioned the difficulty of extracting a spanning tree of high throughput from a given interconnection graph, and we come back to this point in Section 5. 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 Section 3 (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 Bandwidth-Centric Strategy for Tree Shaped Platforms

In this section, we deal with tree-shaped platforms. The root of the tree is always the master. For such platforms, we derive a closed-form expression of the solution of the linear program MSSG(G). We also show that a simple traversal of the tree enables to compute the solution of MSSG(G) in linear time (linear in the size of the platform).

3.1 Star Graph

We start with simple star graphs before dealing with arbitrary tree graphs. A star graph F, as shown in Fig. 4, consists of a node P_0 and its k children $P_1, \ldots, P_k$. In the *full overlap, single-port* model, P_0 can communicate with a single child at a time: It needs c_i units of time to communicate a task to child P_i. Concurrently, P_0 can receive data from its own parent, say P_{-1}, requiring c_0 time per task. We give three examples in Figs. 5, 6, and 7: In the first, all children operate at full rate, and in the latter two, the communication bandwidth is the limiting factor.

Proposition 1. *With the above notations, the minimal value of $n_{\text{task}}(F)$ for the star graph F is obtained as follows:*

1. *Sort the children by increasing communication times. Renumber them so that $c_1 \leq c_2 \ldots \leq c_k$.*
2. *Let p be the largest index so that $\sum_{i=1}^{p} \frac{c_i}{w_i} \leq 1$. If $p < k$, let $\varepsilon = 1 - \sum_{i=1}^{p} \frac{c_i}{w_i}$; otherwise, let $\varepsilon = 0$.*
3. *Then, $n_{\text{task}}(F) = \min\left(\frac{1}{c_0}, \frac{1}{w_0} + \sum_{i=1}^{p} \frac{1}{w_i} + \frac{\varepsilon}{c_{p+1}}\right)$.*

Intuitively, the processors cannot consume more tasks than sent by P_{-1}, hence, the first term of the minimum. For

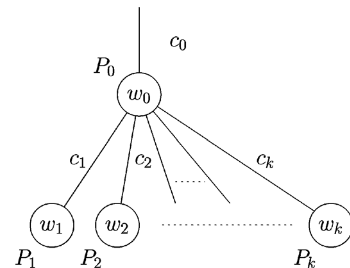

Fig. 4. Star graph.

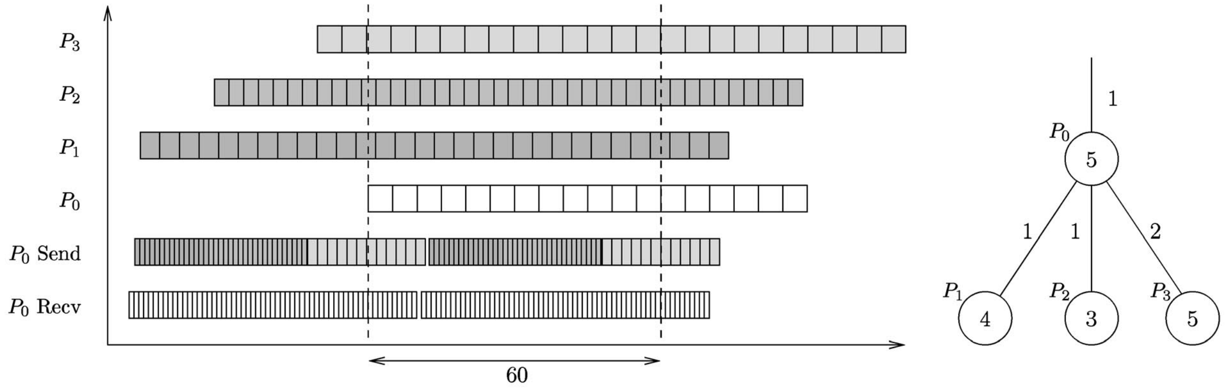

Fig. 5. First example without saturation of the communication bandwidth: All children can be kept fully active.

the second term, when $p = k$, the result is expected: It basically says that children can be fed with tasks fast enough so that they are all kept computing steadily. However, if $p < k$, the result is surprising: In the situation when the communication bandwidth is limited, some children will partially starve: In the optimal solution, these are those with slow communication rates, whatever their processing speeds. In other words, a slow processor with a fast communication link is to be preferred to a fast processor with a slow communication link.

Proof. We use the same notation as in Section 2.3. For $0 \leq i \leq k$, α_i is the fraction of time spent by P_i computing. For $0 \leq i \leq k$, we let s_i (instead of $s_{0,i}$) be the fraction of time spent by P_0 sending tasks to child P_i. Note that we know that tasks flow from P_0 to its children, hence, $r_{0,i} = 0$ (where $r_{0,i}$ is the fraction of time spent by P_0 receiving tasks from P_i). Finally, let r_{-1} be the fraction of time spent by P_0 receiving tasks from its own parent. We have the following linear program:

$$\textbf{Maximize } \frac{\alpha_0}{w_0} + \sum_{i=1}^{k} \frac{\alpha_i}{w_i}$$

$$\textbf{subject to} \begin{cases} 0 \leq \alpha_i \leq 1 \text{ for } 0 \leq i \leq k \\ 0 \leq r_{-1} \leq 1 \\ \sum_{i=1}^{k} s_i \leq 1 \end{cases} \quad \begin{array}{l} \frac{r_{-1}}{c_0} = \frac{\alpha_0}{w_0} + \sum_{i=1}^{k} \frac{s_i}{c_i} \\ \frac{s_i}{c_i} = \frac{\alpha_i}{w_i} \text{ for } 0 \leq i \leq k \end{array}$$

The last equation stands for the conservation equation for each child P_i: Because no task is forwarded, each received task is consumed in place. We normalize these equations by introducing the rates R_i: $R_{-1} = \frac{s_{-1}}{c_0}$ is the rate of tasks per second received from the parent P_{-1}, $R_0 = \frac{\alpha_0}{w_0}$ is the rate they are executed in the parent node P_0, and $R_i = \frac{s_i}{c_i} = \frac{\alpha_i}{w_i}$, for $1 \leq i \leq k$, is the rate they are sent to and executed on the ith child. We obtain the following formulation:

$$\textbf{Base Problem: Maximize } \sum_{i=0}^{k} R_i, \textbf{ subject to}$$

$$(\mathbf{B0})\ R_i \leq \frac{1}{w_i} \text{ for } 0 \leq i \leq k \quad (\mathbf{B2})\ R_i \leq \frac{1}{w_i} \text{ for } 0 \leq i \leq k$$
$$(\mathbf{B1})\ R_{-1} = \sum_{i=0}^{k} R_i \leq \frac{1}{c_0} \quad (\mathbf{B3})\ \sum_{i=1}^{k} R_i c_i \leq 1$$

Let R be the solution of the Base Problem. We claim that $R = \min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right)$ (unless $c_0 = 0$, in that case $R = \frac{1}{w_0} + S$), where S is the solution to the following problem:

$$\textbf{Auxiliary problem: Maximize } \sum_{i=1}^{k} R_i, \textbf{ subject to}$$

$$(\mathbf{1})\ R_i w_i \leq 1 \text{ for } 1 \leq i \leq k, \qquad (\mathbf{2})\ \sum_{i=1}^{k} R_i c_i \leq 1.$$

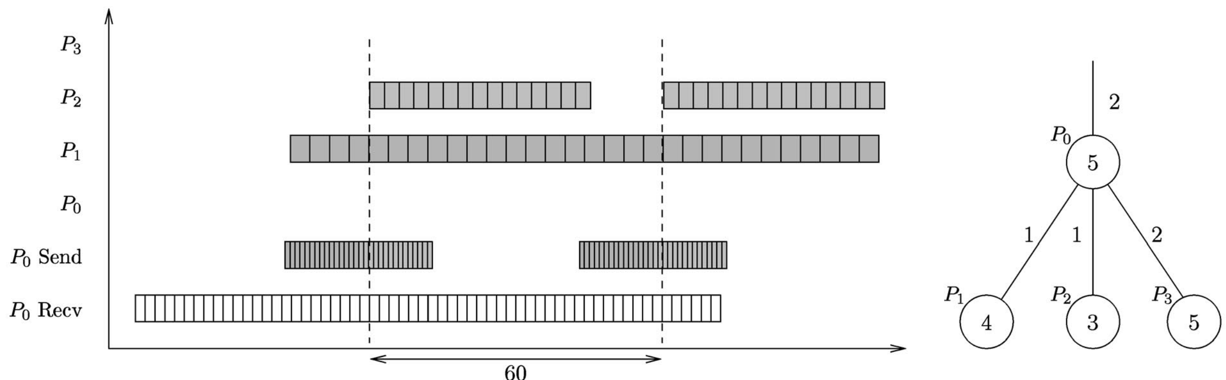

Fig. 6. Second example with saturation of the communication bandwidth: Some children are partially idle due to the low bandwidth between P_0 and its parent.

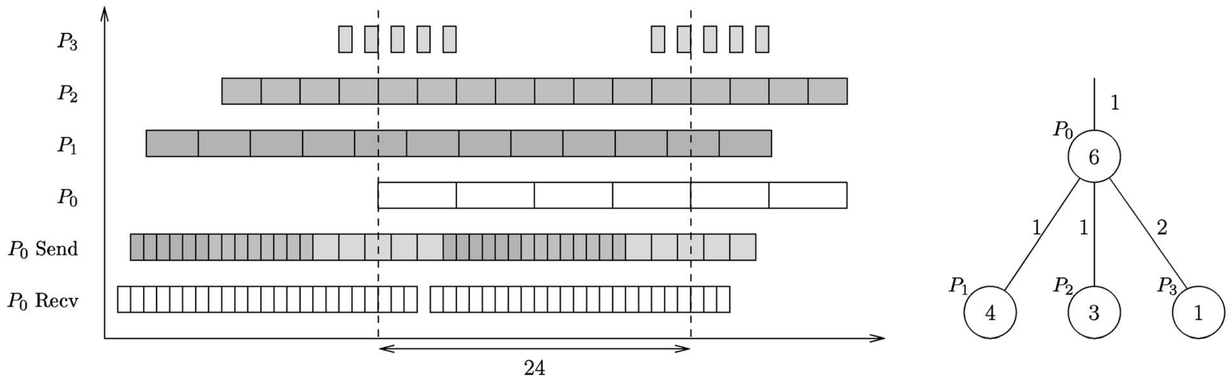

Fig. 7. Third example with saturation of the communication bandwidth: child P_3 is partially idle due to its high computation speed.

Because the auxiliary problem is less constrained than the original one, we immediately have that $\frac{1}{w_0} + S \geq R$. We also have $\frac{1}{c_0} \geq R$ because of (B0) and (B1), hence, $\min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right) \geq R$. To show the reverse inequality, there are two cases, according to the value of $\min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right)$. Assume first that $\min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right) = \frac{1}{c_0}$. Let $(R_1, \ldots, R_k)$ be the optimal solution of the auxiliary problem: $S = \sum_{i=1}^{k} R_i$ and $\frac{1}{w_0} + S \geq \frac{1}{c_0}$. Let $\alpha = \frac{\frac{1}{c_0}}{\frac{1}{w_0}+S} \leq 1$. Then, $\left(\frac{1}{c_0}, \frac{\alpha}{w_0}, \alpha R_1, \ldots, \alpha R_k\right)$ is a solution to the base problem whose objective function is equal to $\frac{1}{c_0}$. Therefore, $\frac{1}{c_0} \leq R$.

Assume now that $\min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right) = \frac{1}{w_0} + S$. Let $(R_1, \ldots, R_k)$ be the optimal solution of the auxiliary problem: $S = \sum_{i=1}^{k} R_i$ and $\frac{1}{w_0} + S \leq \frac{1}{c_0}$. Let $R_0 = \frac{1}{w_0}$ and $R_{-1} = \sum_{i=0}^{k} R_i$. Then, $(R_{-1}, R_0, R_1, \ldots, R_k)$ is a solution to the base problem (note that (B1) is satisfied because of the hypothesis). Hence, $\frac{1}{w_0} + S \leq R$. This concludes the reduction to the auxiliary problem.

We can now come to the solution of the auxiliary problem. As in the statement of the theorem, let p be the largest index so that $\sum_{i=1}^{p} \frac{c_i}{w_i} \leq 1$. Let $R_i^* = \frac{1}{w_i}$ for $1 \leq i \leq p$. If $p < k$, let $R_{p+1}^* = \frac{\varepsilon}{c_{p+1}}$, where $\varepsilon = 1 - \sum_{i=1}^{p} \frac{c_i}{w_i}$. If $p + 1 < k$, let $R_i^* = 0$ for $p + 2 \leq i \leq k$. We claim that (R_i^*) is the optimal solution of the optimization problem:

- First, it is indeed a solution. We have $R_{p+1}^* \leq \frac{1}{w_{p+1}}$ when $p < k$. This comes directly from the definition of p: Since $\sum_{i=1}^{p+1} \frac{c_i}{w_i} > 1$, we have $\varepsilon = 1 - \sum_{i=1}^{p} \frac{c_i}{w_i} < \frac{c_{p+1}}{w_{p+1}}$, hence, $\frac{\varepsilon}{c_{p+1}} \leq \frac{1}{w_{p+1}}$. As for (2), $\sum_{i=1}^{j} R_i c_i = \sum_{i=1}^{j} \frac{c_i}{w_i} \leq 1$ for all $j \leq p$, by definition of p. And. if $p < k$, $\sum_{i=1}^{p+1} R_i c_i = 1$, by definition of R_{p+1}.
- Second, it is the solution that maximizes the objective function. To see this, consider all the optimal solutions, i.e., all the solutions that achieve the optimal value of the objective function. Among these optimal solutions, consider one solution (R_i) such that R_1 is maximal. Assume by contradiction that $R_1 < R_1^* = \frac{1}{w_1}$. Then, there exists at least one index $j > 2$ such that $R_j > R_j^*$; otherwise, the solution would not be optimal. Now, since the c_i are sorted, we have $c_1 \leq c_j$. We do not change the value of the objective function if we let $R_1 = R_1 + \tau$ and $R_j = R_j - \tau$, where τ is an arbitrary small nonnegative rational number. However, we do have a new solution to the optimization problem, because $(R_1 + \tau)c_1 + (R_j - \tau)c_j \leq R_1 c_1 + R_j c_j$. Hence, we have an optimal solution with a larger R_1 than the original one, a contradiction. Hence, we have shown that there exist optimal solutions such that $R_1 = R_1^*$. We restrict to such solutions without loss of generality and we iterate the process: We finally derive that (R_i^*) is an optimal solution.

The optimal solution of the auxiliary problem is $S = \sum_{i=1}^{k} R_i = \sum_{i=1}^{p} \frac{1}{w_i} + \frac{\varepsilon}{c_{p+1}}$.

The optimal solution of the base problem is $R = \min\left(\frac{1}{c_0}, \frac{1}{w_0} + S\right)$, which establishes our claim.

A less formal (but much shorter) proof of Proposition 1 is the following: Sorting the c_i and feeding as many tasks as possible to the children taken in that order maximizes the number of tasks that are communicated to the children; hence, the number of tasks that are processed by the children. Add those processed by the parent, and take the minimum with the input rate to derive the optimal value. Note that the proof in Proposition 1 is fully constructive: The number of tasks to be computed by the parent and to be sent to each child is directly computed from the optimal solution (R_i^*). □

3.2 Arbitrary Tree Graphs

The best allocation of tasks to processor nodes is determined using a bottom-up traversal of the tree:

Proposition 2. *Let T be an arbitrary tree graph. The maximal value of $n_{\text{task}}(F)$ for the whole tree T is obtained as follows:*

1. *Consider any subtree F consisting of several leaves and their parent. Replace this tree with a single node whose weight is $w = \frac{1}{n_{\text{task}}(F)}$, where $n_{\text{task}}(F)$ is given by Proposition 1.*
2. *Iterate the process until there remains a single node.*

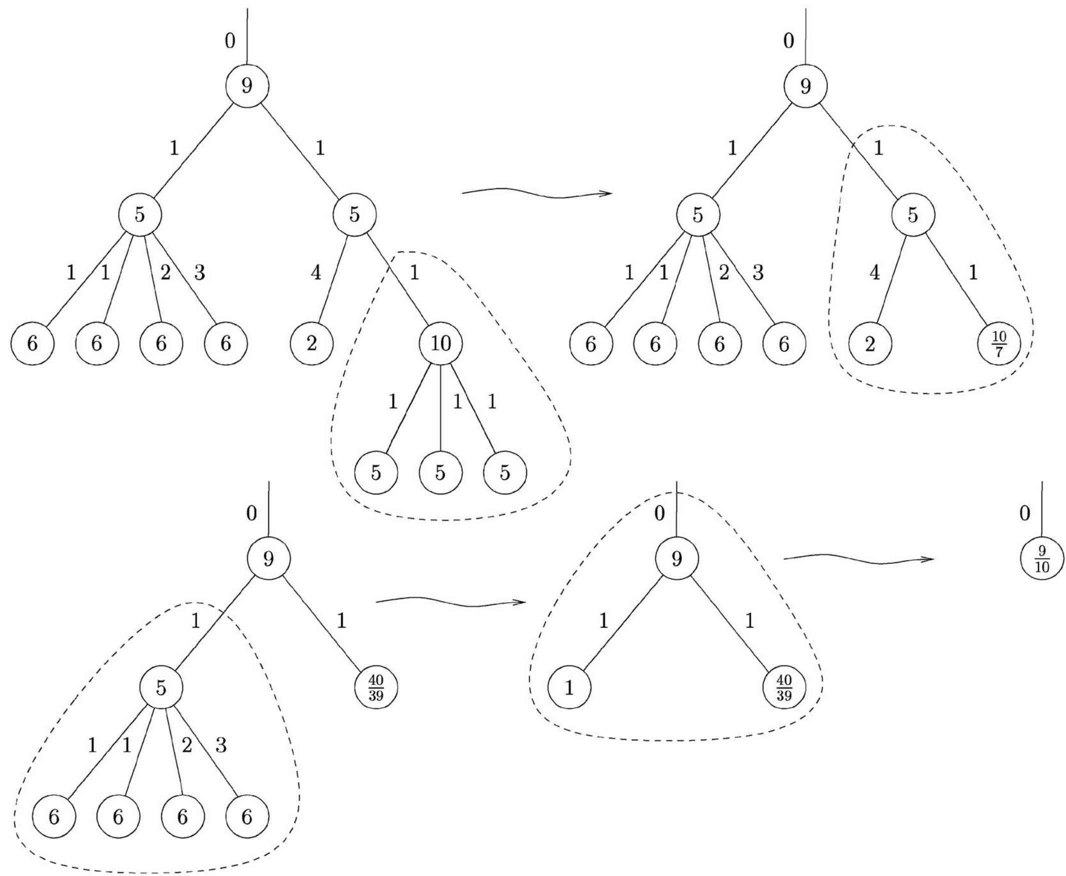

Fig. 8. Computing the best allocation for a tree graph.

Then, the minimal value is equal to the weight of the single node.

Proof. The proof is immediate: In steady state, a star graph F consisting of a parent and several leaves behaves exactly as a single node of weight $w = \frac{1}{n_{\text{task}}(F)}$, where $n_{\text{task}}(F)$ is determined by Proposition 1. For the root node, we can assume a link from a virtual parent with infinite capacity, i.e., $c_0 = 0$.

Again, the proof is fully constructive and the optimal task allocation is computed using the bottom-up approach. Fig. 8 is a small example. □

4 Other Operating Models

In this section, we discuss models of operation other than the full-overlap, single-port model, or base model. For each new model, we show which equations to modify in the linear program MSSG(G). In other words, we perform a series of model reductions to convert each instance of a processor operating a given model into an instance with the base model.

We list different models, starting from the most powerful machines down to purely sequential processors.

$\mathcal{M}(r^*||s^*||w)$: **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 $\mathcal{M}(r||s,w)$ 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.

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 (5) and (6) in the linear program! Indeed, under the new model, (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

$$\forall i, \forall k, 1 \leq k \leq k_1, 0 \leq \sum_{j \in n(i)} r_{ij}^k \leq 1, \quad (9)$$

$$\forall i, \forall k, 1 \leq k \leq k_2, 0 \leq \sum_{j \in n(i)} s_{ij}^k \leq 1. \quad (10)$$

Reduction for $\mathcal{M}(r||s,w)$, the Receive-in-Parallel, single port model. This model is less powerful than the base model: the 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 (5) and (1) by

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

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 (6) by

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

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 (5) and (6) by

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

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 (1), (5), and (6) by

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

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

5 Complexity Results

5.1 Asymptotic Optimality

In this section, we prove that steady state scheduling is asymptotically optimal. Given a platform graph $G = (V, E, w, c)$ and a time bound K, define $opt(G, K)$ as the optimal number of tasks that can be computed using the whole platform, within K time-units. We have the following result.

Lemma 1. $opt(G, K) \leq n_{\text{tasks}}(G) \times K.$

Proof. Consider an optimal scheduling. For each processor P_i, let $t_i(K)$ be the total number of tasks that have been executed by P_i within the K time-units. Similarly, for each edge e_{ij} in the graph, let $t_{i,j}(K)$ be the total number of tasks that have been forwarded by P_i to P_j within the K time-units. The following equations hold true.

- $t_i(K) \cdot w_i \leq K$ (time for P_i to process its tasks).
- $\sum_{j \in n(i)} t_{i,j}(K) \cdot c_{ij} \leq K$ (time for P_i to forward outgoing tasks in the one-port model).
- $\sum_{j \in n(i)} t_{j,i}(K). \cdot c_{ij} \leq K$ (time for P_i to receive incoming tasks in the one-port model).
- $\sum_{j \in n(i)} t_{j,i}(K) = t_i(K) + \sum_{j \in n(i)} t_{i,j}(K)$ (conservation equation).

Let $\alpha_i = \frac{w_i t_i(K)}{K}$, $s_{ij} = \frac{c_{ij} t_{i,j}(K)}{K}$, and $r_{ij} = \frac{c_{ij} t_{j,i}(K)}{K} = s_{ji}$. All the equations of the linear program MSSG(G) hold, hence, $\sum_i \frac{\alpha_i}{w_i} \leq n_{\text{tasks}}(G)$, the optimal value. Going back to the original variables, we derive:

$$opt(G, K) = \sum_i t_i(K) \leq n_{\text{tasks}}(G) \times K.$$

□

Basically, Lemma 1 says that no scheduling can execute more tasks than the steady state. There remains to bound the initialization and the clean-up phase to come up with a well-defined scheduling algorithm based upon steady state operation. Consider the following algorithm (assume K is large enough):

- Solve the linear program MSSG(G): Compute the maximal throughput $n_{\text{tasks}}(G)$, compute all the values α_i, r_{ij}, and r_{ij}, and determine the time-period T. For each processor P_i, determine per_i, the total number of tasks that it receives per period. Note that all these quantities are independent of K: They depend only upon the characteristics w_i and c_{ij} of the platform graph.
- Initialization: The master sends per_i tasks to each processor P_i. This requires I units of time, where I is a constant independent of K.
- Let J be the maximum time for each processor to consume per_i tasks ($J = \max_i \{per_i.w_i\}$). Again, J is a constant independent of K.
- Let $r = \lfloor \frac{K-I-J}{T} \rfloor$.
- Steady state scheduling: During r periods of time T, operate the platform in steady state, according to the solution of MSSG(G).

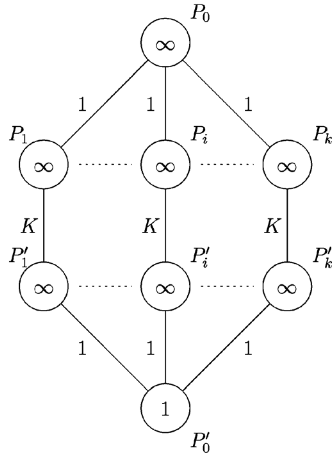

Fig. 9. Graph G used for the proof of the inapproximability.

- Clean-up: Do not forward any task, but consume in place all remaining tasks. This requires at most J time-units. Do nothing during the very last units ($K - I - J$ may not be evenly divisible by T).
- The number of tasks processed by this algorithm within K time-units is equal to $steady(G, K) = (r+1) \times T \times n_{\text{tasks}}(G)$.

Proposition 3. *The previous scheduling algorithm based upon steady state operation is asymptotically optimal:*

$$\lim_{K \to +\infty} \frac{steady(G, K)}{opt(G, K)} = 1.$$

Proof. Using Lemma 1, $opt(G, K) \leq n_{\text{tasks}}(G).K$. From the description of the algorithm, we have $steady(G, K) = ((r+1)T).n_{\text{tasks}}(G) \geq (K - I - J).n_{\text{tasks}}(G)$, whence the result because I, J, T, and $n_{\text{tasks}}(G)$ are constants independent of K. □

5.2 Spanning Trees

For a general interconnection graph, the solution of the linear programming problem may lead to the use of multiple paths (this is the case for the toy example of Section 2.4). 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 [4] 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 (probably exponential) cost to determine the best tree, there exist graphs for which the throughput of the best tree is arbitrarily bad compared to the throughput that can be achieved while the whole graph.

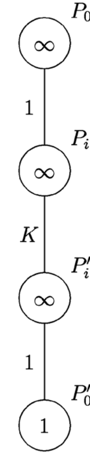

Fig. 10. Description of a tree T extracted from G.

5.2.1 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:

Defintion 1 (BEST-TREE(G)). *Let $G = (V, E, w, c)$ be the node-weighted edge-weighted graph representing the architectural framework. Find the tree $T = (V, E', w, c)$ that is a subgraph of G rooted at the master, such that the number of tasks $n_{\text{task}}(T)$ 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 $T = (V, E', w, c)$ that is a subgraph 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) is proven to be NP-complete in [3] by reduction from 2-PARTITION, which is known to be NP-Complete [9]. □

5.2.2 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 that is a subgraph of G and rooted at the master, we have*

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

Proof. Consider the graph depicted in Fig. 9.

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 Fig. 10 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,

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

□

6 Related Problems

We classify several related papers along the following lines:

Scheduling task graphs on heterogeneous platforms. Several heuristics have been introduced to schedule (acyclic) task graphs on different-speed processors, see [21], 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 [12], [19] suggest taking communication contention into account.

Scheduling divisible load. Instead of scheduling several communications, one for each task, the divisible load approach consists in scheduling a single communication at the beginning of the operation. The cost of this communication is proportional to the amount of computation performed. A star graph is targeted in [20], with homogeneous links and different-speed processors. The extension to heterogeneous links is dealt with in [6]. See also [14], [1] for more results on sharing bag of tasks in heterogeneous clusters.

Master-slave on the computational grid. Master-slave scheduling on the grid can be based on a network-flow approach [17] or on an adaptive strategy [11]. Note that the network-flow approach of [17] 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 [10], [22].

7 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. We derive from this steady-state strategy a asymptotically optimal schedule. We have also given a closed-form expression and a more efficient algorithm (linear in the processor number) for tree-shaped platform graphs using a *bandwidth-centric* approach. Some simulation results on demand-driven heuristics based on the *bandwidth-centric* approach are given in [4] and [13].

We have also 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. Nevertheless, several low-cost heuristics that achieve very good performances on a wide range of simulations are proposed in [2]. 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 real-life problems.

Acknowledgments

The authors would like to thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper.

References

[1] M. Adler, Y. Gong, and A.L. Rosenberg, "Optimal Sharing of Bags of Tasks in Heterogeneous Clusters," *Proc. 15th ACM Symp. Parallelism in Algorithms and Architectures*, pp. 1-10, 2003.

[2] C. Banino, O. Beaumont, A. Legrand, and Y. Robert, "Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids," *Proc. Int'l Conf. Applied Parallel Computing*, pp. 423-432, 2002.

[3] C. Banino, O. Beaumont, A. Legrand, and Y. Robert, "Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Grids," Technical Report 2002-12, LIP, Mar. 2002.

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms," *Proc. Int'l Parallel and Distributed Processing Symp.*, 2002.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Optimal Algorithms for the Pipelined Scheduling of Task Graphs on Heterogeneous Systems," Technical Report RR-2003-29, LIP, ENS Lyon, France, May 2003.

[6] S. Charcranoon, T.G. Robertazzi, and S. Luryi, "Optimizing Computing Costs Using Divisible Load Analysis," *IEEE Trans. Computers*, vol. 49, no. 9, pp. 987-991, Sept. 2000.

[7] *Scheduling Theory and Its Applications.* P. Chrétienne, E.G. Coffman Jr., J.K. Lenstra, and Z. Liu, eds. John Wiley and Sons, 1995.

[8] Entropia, http://www.entropia.com, 2003.

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

[10] J.P Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," *Proc. Ninth IEEE Int'l Symp. High Performance Distributed Computing*, 2000.

[11] E. Heymann, M.A. Senar, E. Luque, and M. Livny, "Adaptive Scheduling for Master-Worker Applications on the Computational Grid," R. Buyya and M. Baker, eds., *Proc. Workshop Grid Computing*, pp. 214-227, 2000.

[12] T.S. Hsu, J.C. Lee, D.R. Lopez, and W.A. Royce, "Task Allocation on a Network of Processors," *IEEE Trans. Computers*, vol. 49, no. 12, pp. 1339-1353, 2000.

[13] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "Autonomous Protocols for Bandwidth-Centric Scheduling of Independent Task Applications," *Proc. Int'l Parallel and Distributed Processing Symp.*, 2003.

[14] A.L. Rosenberg, "On Sharing Bags of Tasks in Heterogeneous Networks of Workstations: Greedier is Not Better," *Proc. Int'l Conf. Cluster Computing*, pp. 124-131, 2001.

[15] A. Schrijver, "Combinatorial Optimization: Polyhedra and Efficiency," *Algorithms and Combinatorics*, vol. 24, Springer-Verlag, 2003.

[16] SETI, http://setiathome.ssl.berkeley.edu, 2003.

[17] G. Shao, F. Berman, and R. Wolski, "Master/Slave Computing on the Grid," *Proc. Heterogeneous Computing Workshop*, 2000.

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

[19] O. Sinnen and L. Sousa, "Comparison of Contention-Aware List Scheduling Heuristics for Cluster Computing," *Proc. Workshop Scheduling and Resource Management for Cluster Computing*, pp. 382-387, 2001.

[20] J. Sohn, T.G. Robertazzi, and S. Luryi, "Optimizing Computing Costs Using Divisible Load Analysis," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 225-234, Mar. 1998.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task Scheduling Algorithms for Heterogeneous Processors," *Proc. Eighth Heterogeneous Computing Workshop*, 1999.

[22] J.B. Weissman, "Scheduling Multi-Component Applications in Heterogeneous Wide-Area Networks," *Proc. Heterogeneous Computing Workshop*, 2000.

Cyril Banino is currently a PhD student in NTNU, the Norvegian University of Science and Technology. He is mainly interested in high-performance scientific computing and in combinatorial optimization.

Olivier Beaumont received the 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.

Larry Carter received the AB degree from Dartmouth College in 1969 and the PhD degree in mathematics from the University of California at Berkeley in 1974. He worked at as a research staff member and manager at IBM's T.J. Watson Research Center for nearly 20 years in the areas of probabilistic algorithms, compilers, VLSI testing, and high-performance computation. Since 1994, Dr. Carter has been a professor in the Computer Science and Engineering Department of the University of California at San Diego. Between 1996 and 2000, he served as vice chair and then chair of the department. His current research interests include scientific computation, performance programming, parallel computation, and computer architecture. Prof. Carter is a senior fellow at the San Diego Supercomputing Center and a fellow of the IEEE.

Jeanne Ferrante received the PhD degree in mathematics from MIT in 1974. She was a research staff member at the IBM T.J. Watson Research Center from 1978 to 1994, and currently is a professor of computer science and associate dean of the Jacobs School of Engineering at the University of California, San Diego. Her work has included the development of intermediate representations for optimizing and parallelizing compilers, most notably the Program Dependence Graph and Static Single Assignment form. Her interests also include optimizing for parallelism and memory hierarchy and scheduling on large distributed platforms. She is a fellow of the ACM and a senior member of the IEEE.

Arnaud Legrand 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 techniques.

Yves Robert received the 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 are 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 Transactions on Parallel and Distibuted Systems*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.