Centralized versus Distributed Schedulers for Bag-of-Tasks Applications

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

Abstract—Multiple applications that execute concurrently on heterogeneous platforms compete for CPU and network resources. In this paper, we consider the problem of scheduling applications to ensure fair and efficient execution on a distributed network of processors. We limit our study to the case where communication is restricted to a tree embedded in the network, and the applications consist of a large number of independent tasks (Bags of Tasks) that originate at the tree's root. The tasks of a given application all have the same computation and communication requirements, but these requirements can vary for different applications. The goal of scheduling is to maximize the throughput of each application while ensuring a fair sharing of resources between applications. We can find the optimal asymptotic rates by solving a linear programming problem that expresses all necessary problem constraints, and we show how to construct a periodic schedule from any linear program solution. For single-level trees, the solution is characterized by processing tasks with larger communication-to-computation ratios at children with larger bandwidths. For multilevel trees, this approach requires *global* knowledge of all application and platform parameters. For large-scale platforms, such global coordination by a centralized scheduler may be unrealistic. Thus, we also investigate decentralized schedulers that use *only local* information at each participating resource. We assess their performance via simulation and compare to an optimal centralized solution obtained via linear programming. The best of our decentralized heuristics achieves the same performance on about 2/3 of our test cases but is far worse in a few cases. Although our results are based on simple assumptions and do not explore all parameters (such as the maximum number of tasks that can be held on a node), they provide insight into the important question of fairly and optimally scheduling heterogeneous applications on heterogeneous grids.

Index Terms—Parallel computing, scheduling, multiple applications, bag of tasks, resource sharing, fairness, throughput.

1 Introduction

In this paper, we consider the problem of scheduling multiple applications that are executed concurrently and, hence, that compete for CPU and network resources. The target computing platform is a heterogeneous network of computers structured either as a *star network* (a one-level rooted tree) or a multilevel rooted tree. In both cases, we assume full heterogeneity of the resources, both for CPU speeds and link bandwidths.

Each application consists of a large collection of independent equally sized tasks, and all tasks originate at the tree's root. This scenario is somewhat similar to that addressed by existing systems. For instance, BOINC [1] is a centralized scheduler that distributes tasks for participating applications such as SETI@home, ClimatePrediction.NET, and Einstein@Home. The applications can be very different in nature, for example, files to be processed, images to be analyzed, or matrices to be manipulated. Consequently, we assume each application has an associated *communication-to-computation ratio* (CCR) for all of its tasks. This ratio proves to be an important parameter in the scheduling process.

The scheduling problem is to maintain a balanced execution of all applications while using the computational and communication resources of the system effectively to maximize the throughput $\alpha^{(k)}$ of each application (the average number of tasks of application A_k, $1 \leq k \leq K$, processed per time unit). For each application, the root node must decide which workers (that is, which subtree) the tasks are sent to. For multilevel trees, each nonleaf worker must make similar decisions: which tasks to compute locally and which to forward to workers further down in the tree. The scheduler must also ensure a fair management of the resources. If all tasks were equally important, the scheduler should aim to process the same number of tasks for each application. We could generalize this by allowing each application A_k to be assigned a *priority weight* $w^{(k)}$ that quantifies its relative value. For instance, if $w^{(1)} = 3$ and $w^{(2)} = 1$, the scheduler should try to ensure that three tasks of A_1 are executed for each task of A_2. However, we avoid using weights in the following to simplify the presentation.[1]

We will consider both centralized and decentralized schedulers. For smaller platforms, it may be realistic to

- *O. Beaumont is with Laboratoire LaBRI, CNRS-INRIA, domaine Universitaire, 351 cours de la Libération, 33405 Talence, France. E-mail: Olivier.Beaumont@labri.fr.*
- *L. Carter and J. Ferrante are with the Department of Computer Science and Engineering, University of California, San Diego, 9800 Gilman Drive, Mail Code 0404, La Jolla, CA 92093-0404. E-mail: {carter, ferrante}@cs.ucsd.edu.*
- *A. Legrand is with CNRS, Laboratoire LIG, 51 Avenue Jean Kuntzmann, 38330 Montbonnot St., Martin, France. E-mail: Arnaud.Legrand@imag.fr.*
- *L. Marchal and Y. Robert are with Laboratoire LIP, ENS LYON, CNRS-INRIA, École Normale Supérieure de Lyon, 46 allee d;Italie, F-69364 Lyon Cedex 07, France. E-mail: {Loris.Marchal, Yves.Robert}@ens-lyon.fr.*

Manuscript received 30 June 2006; revised 7 May 2007; accepted 5 June 2007; published online 27 July 2007.
Recommended for acceptance by D. Trystram.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0174-0606.
Digital Object Identifier no. 10.1109/TPDS.2007.70747.

1. All results can be easily extended when adding weights: simply replace $\alpha^{(k)}$ by $\frac{\alpha^{(k)}}{w^{(k)}}$. See [2] for further details.

assume a centralized scheduler, which makes its decisions based upon complete and reliable knowledge of all application and platform parameters. With such knowledge at our disposal, we are able to determine an *optimal* schedule, that is, a schedule that maximizes the fair throughput asymptotically. This is done by formulating all constraints into a linear programming problem and using the solution to construct a periodic schedule. Except during the (fixed-length) start-up and clean-up periods, no schedule can have a higher throughput. For single-level rooted trees, we provide an interesting characterization of the optimal solution: applications with a larger CCR should be processed by the workers with larger bandwidths, independently of the CCRs of the workers.

For large-scale platforms, particularly ones in which resource availability changes over time, a centralized scheduler may be undesirable. Only local information such as the current capacity (CPU speed and link bandwidth) of a processor's neighbors is likely to be available. One major goal of this paper is to investigate whether decentralized scheduling algorithms can reach an optimal throughput or at least achieve a significant fraction of it. We provide several decentralized heuristics that rely exclusively on local information to make scheduling decisions. The key underlying principles of these heuristics come from our characterization of the optimal solution for star networks: give priority to high-bandwidth children and assign them tasks of larger CCRs. We evaluate the decentralized heuristics through extensive simulations using SimGrid [3] and use a centralized algorithm (guided by the linear program solution) as a reference basis.

The rest of the paper is organized as follows: Section 2 is devoted to an overview of related work. In Section 3, we state precisely the scheduling problem under consideration, with all application and platform parameters, and discuss the objective function used afterward. Section 4 explains how to analytically compute the best solution, using a linear programming approach, and characterizes the solution for single-level trees. Section 5 is then a discussion on the design of several decentralized scheduling heuristics, whereas Section 6 provides an experimental comparison of these heuristics. Finally, we state some concluding remarks in Section 7.

2 Related Work

We classify related research in three main areas:

2.1 Bag-of-Tasks Scheduling on Computational Grids

Bag-of-Tasks applications are parallel applications whose tasks are all independent. This framework is motivated by problems that are addressed by collaborative computing efforts such as SETI@home [4], factoring large numbers [5], the Mersenne prime search [6], and those distributed computing problems organized by companies such as Entropia [7]. One can argue that Bag-of-Tasks applications are most suited for computational grids, because communication can easily become a bottleneck for tightly coupled parallel applications.

Condor [8] and APST [9], [10] are one of the first projects providing specific support for such applications. Condor was initially conceived for campus-wide networks [8] but has been extended to run on grids [11]. Whereas APST is user centric and does not handle multiple applications, Condor is system centric. Those two projects are designed for standard grids, but more recent and active projects like OurGrid [12] and BOINC [1] target more distributed architectures like desktop grids. BOINC [1] is a centralized scheduler that distributes tasks for participating applications such as SETI@home, ClimatePrediction.NET, and Einstein@Home. The set of resources is thus very large, whereas the set of applications is small and very controlled. OurGrid is a Brazilian project that encourages people to donate their computing resources while maintaining the symmetry between consumers and providers. Whereas APST, Condor, and BOINC all rely on a centralized scheduler and may suffer from scalability issues, OurGrid is the only framework we know of where scheduling is done in a fully distributed way, following the peer-to-peer approach.

However, all these projects generally focus on designing and providing a working infrastructure, and they do not provide any analysis of scheduling techniques suited to such environments.

2.2 Steady-State Scheduling

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 a distributed platform?), it makes sense to focus on *steady-state* optimization problems rather than on standard *makespan* minimization problems. Minimizing the makespan, that is, the total execution time, is an NP-hard problem in most practical situations [13], [14], [15], whereas it turns out that the optimal steady state can often be characterized very efficiently, with low-degree polynomial complexity.

The steady-state approach has been pioneered by Bertsimas and Gamarnik [16]. It has been used successfully in many situations [17]. In particular, steady-state scheduling has been used to schedule independent tasks on heterogeneous tree-overlay networks [18], [19]. This is the same problem dealt with in the present paper but restricted to a single application. Bandwidth-centric scheduling is introduced in [18], and extensive experiments are reported in [20]. Autonomous protocols for bandwidth-centric scheduling are investigated by Carter et al. [21]. Such distributed autonomous protocols have been obtained only on tree platforms. That is why in the current more complex context, we restrict our study to tree platforms. The steady-state approach has also been used by Hong and Prasanna [22] who extend the work in [18] to deploy a divisible workload on a heterogeneous platform. However, to the best of our knowledge, the steady-state scheduling approach has never been used in a multiple-application context.

2.3 Fairness

In a multiuser environment, resources have to be fairly shared between users. This issue becomes more and more critical as the size of the system increases. There is actually a

large gap between what is known on the theoretical side in game theory and what is implemented in practical Bag-of-Tasks scheduling environments. In most practical environments, some fairness is ensured through the use of hand-tuned priorities or reciprocation-based economy [23]. Market-inspired economy and an auction-based mechanism could also be used. However, in most existing work related to fairness in grid environments, the mechanism is not based on clear definitions of the fairness criteria.

Fairness is yet a classical criterion in network bandwidth allocation. Optimizing the sum of the throughputs is known as maximizing the throughput or *profit* of a network. Optimizing this kind of objective is natural for an access provider who receives an amount of money proportional to the throughput that he/she is able to provide and who wants to maximize his/her profit. However, this criterion is known to be unfair and can lead to starvation. That is why in Section 3.4, we choose to maximize the minimum of $\alpha^{(k)}$ rather than the sum. This criterion is known in the literature as *max-min* and is intuitively fair since all throughputs are computed to be as close as possible from each other. Between these two extremes, other criteria can be found (for example, *proportional fairness* that maximizes $\sum \log(\alpha^{(k)})$ and *minimum potential delay* that minimizes $\sum \frac{1}{\alpha^{(k)}}$). In fact, all these criteria (profit, proportional fairness, and minimum potential delay) amount to maximizing the arithmetic, geometric, and harmonic mean of the throughput [24]. It is well known in the networking community [25] that max-min fairness is generally achieved by explicit rate calculation (for example, in ATM networks) and rather hard to achieve in a fully decentralized way. Nevertheless, fully distributed algorithms are known to realize proportional fairness (such as some variants of TCP). Adapting such algorithms to Bag-of-Tasks scheduling environments seems challenging as both communications and computations are involved.

3 Framework and Models

In this section, we clarify the assumptions underlying our work; although they are overly simplistic, we believe nevertheless that they provide insight into the important question of how to optimally and fairly schedule heterogeneous applications on heterogeneous grids.

3.1 Platform Model

The target computing platform is either a single-level tree (also called a *star network*) or an arbitrary tree. The processor at the root of the tree is denoted P_0. There are P additional "worker nodes" $P_1, P_2, \ldots, P_P$; each worker P_u has a single parent $P_{p(u)}$, and the link between P_u and its parent has bandwidth b_u. We assume a linear-cost communication model; hence, it takes X/b_u time units to send a message of size X from $P_{p(u)}$ to P_u. For the sake of simplicity, we ignore processor-task affinities; instead, we assume that only the number of floating-point operations per second (c_u for processor P_u) determines the application execution speed.

There are several scenarios for the operation of the processors. In this paper, we concentrate on the *full-overlap single-port model* [26], [27]. 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, there are at most two communications involving a given processor, one sent and the other received.

3.2 Application Model

We consider K applications, A_k, $1 \leq k \leq K$. The root node P_0 initially holds all the input data necessary for each application A_k. Each application is composed of a set of independent equally sized tasks. We can think of each A_k as a bag of tasks, and the tasks are files that require some processing. A task of application A_k is called a task of *type* k. We assume that one can express the computational requirements of tasks as a number of floating-point operations, and we let $c^{(k)}$ be the amount of computation (in floating-point operations) required to process a task of type k. Similarly, $b^{(k)}$ is the size (in bytes) of (the file associated to) a task of type k. We assume that the only communication required is outward from the root, that is, that the amount of data returned by the worker is negligible. Our results are equally applicable to the scenario in which the input to each task is negligible but the output is large. Note that our notations use subscripts for platform resources (bandwidth b_u and CPU speed c_u) and superscripts for application parameters (bytes $b^{(k)}$ and floating-point operations $c^{(k)}$).

3.3 Steady-State Scheduling

If each application had an unlimited supply of tasks, each application should aim at maximizing its average number of task processed per time unit (the *throughput*). When the number of tasks is very large, optimizing the steady-state throughput enables to derive periodic asymptotically optimal schedules for the makespan [18], [19]. In our setting where each application has a very large number of tasks, we should thus try to optimize the steady-state throughput of each application.

More formally, for a given infinite schedule, we can define $N^{(k)}(t)$ the number of tasks of type k processed in time interval $[0, t]$. The throughput for application k of such a schedule is defined as $\alpha^{(k)} = \liminf_{t\to\infty} \frac{N^{(k)}(t)}{t}$. Similarly, we can define

- $\alpha_u^{(k)}$ as the average number of tasks of type k executed by P_u per time unit and
- $sent_{u\to v}^{(k)}$ as the average the number of tasks of type k received by P_v from P_u per time unit.

We will see in Section 4.1 that the $\alpha_u^{(k)}$'s and the $sent_{u\to v}^{(k)}$'s are linked by linear equations and satisfy linear constraints, which enables the derivation of upper bounds on the throughput. It is possible to build a *periodic schedule*, that is, a schedule that repeatedly begins and ends in the same state (see [18] and [19] for more details), from the values of $\alpha_u^{(k)}$ and $sent_{u\to v}^{(k)}$ returned by the linear program.

The throughput achieved by this periodic schedule for each application k is optimal. In other words, when the number of tasks per application is large, this approach enables circumventing the NP-completeness of the makespan optimization problem while deriving efficient schedules.

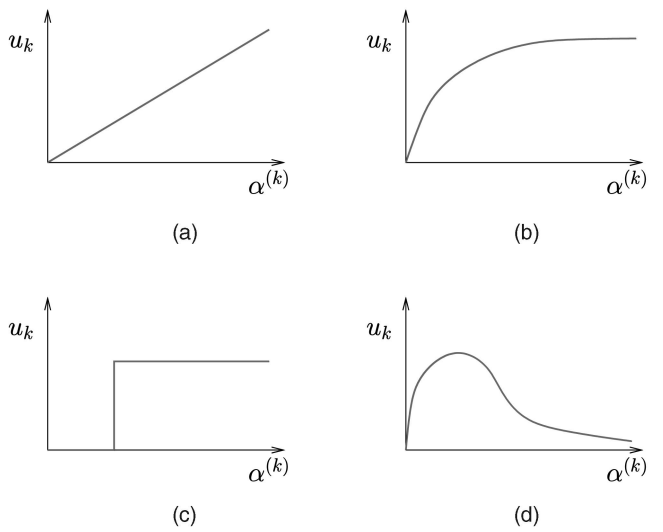

Fig. 1. Examples of utility functions. (a) Linear utility function. (b) Voice-over-IP utility function. (c) Threshold utility function. (d) Price-accounting utility function.

3.4 Objective Function

In this section, we present a few game-theoretic notions and how they translate to our context. This helps us to correctly define a metric to optimize in our setting. Game theory provides a general framework to model situations where many users compete for resources. Each user (in our context, each application) is characterized by a *utility* function u_k defined on $(\alpha_p^{(k)})_{1\leq k\leq K, 1\leq p\leq P}$, where K is the number of applications, and P is the number of computing resources. A variety of possible utility functions are shown in Fig. 1; in the rest of the article, we focus on linear utility functions:

$$u_k(\alpha) = \sum_p \alpha_p^{(k)} = \alpha^{(k)}.$$

Our goal is to find scheduling strategies such that the utility of *each* user is maximized. However, as these users may compete for the same resources, it is generally not possible to simultaneously maximize the utility of each user. Instead, we employ a utility set U:

$$U = \{(u_1(\alpha), \ldots, u_K(\alpha)) | \alpha \text{ is feasible}\}.$$

For tree-shaped platforms, the set of constraints on $\alpha_u^{(k)}$ is a set of linear inequalities (as seen later in Section 4.1), and we know that the utility set is thus a convex polyhedron. Using the same techniques as in [28], [17], and [29], one can show for general platforms that the utility set is also a convex polyhedron, as illustrated in Fig. 2.

Fig. 2a corresponds to the typical situation where two applications are competing on a single node:

$$\begin{cases} \alpha^{(1)} \cdot c^{(1)} + \alpha^{(2)} \cdot c^{(2)} \leq c_u & \alpha^{(1)} \geq 0, \\ \alpha^{(1)} \cdot b^{(1)} + \alpha^{(2)} \cdot b^{(2)} \leq b_u & \alpha^{(2)} \geq 0. \end{cases}$$

In a multiuser context, optimality is not defined as simply as in the single-user context, and it is common to use Pareto optimality, defined as follows:

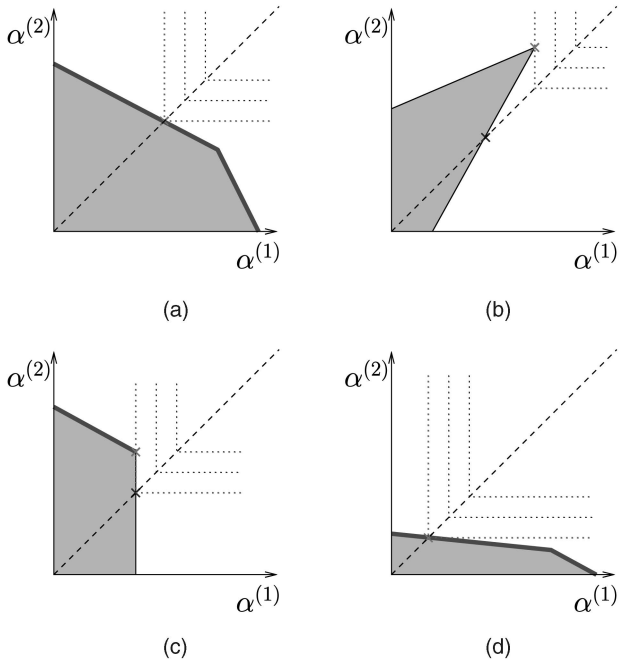

Fig. 2. A few examples of utility sets. The dotted lines are isolines of $(\alpha^{(1)}, \alpha^{(2)}) \to \min(\alpha^{(1)}, \alpha^{(2)})$, and the bold lines represent Pareto optimal points. (a) Conflict on a worker. (b) Synergy. (c) Independency. (d) Typical utility set for a tree.

Definition 1 (Pareto optimality). *$\widetilde{\alpha}$ is Pareto optimal if and only if*

$$\forall \alpha, \exists i, u_i(\alpha) > u_i(\tilde{\alpha}) \Rightarrow \exists j, u_j(\alpha) < u_j(\tilde{\alpha}).$$

In other words, $\widetilde{\alpha}$ is Pareto optimal if it is impossible to strictly increase the utility of a user without strictly decreasing that of another. For example, in Fig. 2a, all points on the rightmost boundaries are Pareto optimal, whereas in Fig. 2b, only the rightmost point is Pareto optimal. Any Pareto-optimal point is thus a priori as worth of interest as any other Pareto-optimal point. Defining fairness can be seen as defining a criterion for choosing among Pareto-optimal points.

One of the most common fairness criterion is the well-known max-min fairness strategy [30], [25]. For a given allocation α, there is an application k whose utility is smaller than the other ones. A max-min fair allocation is such that this smaller utility is as large as possible. Such a criterion is thus more reasonably fair than, for example, trying to maximize the sum of utilities (also known as social optimum). Indeed, with such a criterion, some applications may receive nothing at all (for example, in Fig. 2d). Moreover, by using a weighted throughput, any Pareto-optimal point is a max-min fair point for a given set of weights. We will thus focus in the following on max-min fairness:

$$\text{Maximize} \min_k \alpha^{(k)}.$$

In Fig. 2a, the minimum of the $\alpha^{(k)}$'s is maximized at a point where all the $\alpha^{(k)}$'s have the same value. One can easily check that the only Pareto-optimal point in Fig. 2b is also the point such that the minimum of the $\alpha^{(k)}$'s is maximized. However, one can also check that the points such that the $\alpha^{(k)}$'s all have

the same value are not efficient, which corresponds to the well-known fact that giving the same thing to each user is not always a good option. In fact, the shape of this utility set is rather uncommon and corresponds to a situation where there is a synergy between both users. Such situations may occur with caching mechanisms, for example, but cannot occur in our framework because all coefficients of the linear constraints are positive.

One may then wonder whether in our context, max-min optimal solutions are always such that the throughputs of all applications are the same or not. We will see now that this is true on trees but does not hold on general platforms. The utility set in Fig. 2c is typical of the case where two applications originate from different locations and where one of them can only use a limited area of the network (due to a very high CCR and a small connectivity to the network, for example). In such case, it may be possible to increase the throughput of the application with a lower ratio ($\alpha^{(2)}$ here) without decreasing the throughput of the higher one ($\alpha^{(1)}$ here). However, if both applications start using the same resources, the throughput of one application can only increase at the expense of the throughput of another application. It is important to note that many different points maximize the minimum of the throughputs (all points belonging to U and to the lowest isoline of $\min(\alpha^{(1)}, \alpha^{(2)})$). However, only one of them is of interest (that is, Pareto optimal). It is well known in the network community (see, for example, [30] and [25]) that max-min fairness should be *recursively* defined in this case: the first minimum should be maximized, then the second should be maximized, and so on.

Such situations cannot occur on tree-shaped platforms as applications originate from the same location and thus always compete on the same set of resources. Note that this result does not only hold for the mentioned full-overlap single-port model but applies to any situation where applications originate from the same location. That is why in the rest of this article, we can search for solutions where all application throughputs are equal. However, in a more general situation, we should look for Pareto-optimal allocations, and the previous stopping condition could not be used anymore.

4 Computing the Optimal Solution

In this section, we show how to describe the optimal throughput using a linear programming formulation. For star networks, we give a nice characterization of the solution, which will guide the design of some heuristics in Section 5.

4.1 Linear Programming Solution

A summary of our notation is given as follows:

- P_0 is the root processor, and $P_{p(u)}$ is the parent of node P_u for $u \neq 0$.
- $\Gamma(u)$ is the set of indices of the children of node P_u.
- Node P_u can compute c_u floating-point operations per time unit, and if $u \neq 0$, it can receive b_u bytes from its parent $P_{p(u)}$.
- Each task of type k involves $b^{(k)}$ bytes and $c^{(k)}$ floating-point operations.

The linear programming formulation in (1) allows us to solve for the following variables:

- $\alpha_u^{(k)}$, the average number of tasks of type k executed by P_u per time unit on the whole platform,
- $\alpha^{(k)}$, the average number of tasks of type k executed per time unit, and
- $sent_{u\to v}^{(k)}$, the average number of tasks of type k received by P_v from P_u per time unit.

Any valid schedule must satisfy the linear constraints of (1); we seek a schedule also satisfying the optimal value of the objective function:

$$\text{Maximize } \min_k\left\{\alpha^{(k)}\right\} \text{ under the constraints}$$

$$\begin{cases} \forall k, & \sum_u \alpha_u^{(k)} = \alpha^{(k)} \text{ (definition of } \alpha^{(k)}\text{)}, \\ \forall k, \forall u \neq 0, & sent_{p(u)\to u}^{(k)} = \alpha_u^{(k)} + \sum_{v\in\Gamma(u)} sent_{u\to v}^{(k)} \\ & \text{(data conservation)}, \\ \forall u, & \sum_k \alpha_u^{(k)} \cdot c^{(k)} \leq c_u \\ & \text{(computation limit at node } P_u\text{)}, \\ \forall u, & \sum_{v\in\Gamma(u)} \frac{\sum_k sent_{u\to v}^{(k)} \cdot b^{(k)}}{b_v} \leq 1 \\ & \text{(communication limit out of } P_u\text{)}, \\ \forall k, u & \alpha_u^{(k)} \geq 0 \text{ and } sent_{u\to v}^{(k)} \geq 0 \text{ (nonnegativity)}. \end{cases} \tag{1}$$

All the input parameters to the linear programming problem are rational numbers, and the solution will be rational also (and, hence, computed in polynomial time).

As explained in Section 3.4, on tree-shaped platforms, the previous solution is the max-min fair solution. On the general platform, similar constraints can be written, but solving this linear program would not give the max-min fair solution as the first minimum should be maximized, then the second should be maximized, and so on. This can easily be done in our setting by identifying which applications correspond to the first minimum by looking at saturated constraints (those inequalities that are in fact equalities at optimum point). One can then rerun the linear program, with the throughput of these applications fixed, to maximize the smallest throughput of the remaining applications. This process can be repeated until all applications are saturated. Max-min solutions can thus easily be computed in polynomial time, even on complex platforms.

4.2 Reconstructing a Periodic Schedule from a Linear Programming Solution

The linear inequalities in the linear programming problem (1) describe steady-state behavior, but it is not immediately obvious that there exists a valid schedule satisfying these constraints that also achieves the desired throughput. Nevertheless, let us suppose that such a solution exists, and we have determined all the values $\alpha_u^{(k)}$ and $sent_{u\to v}^{(k)}$. We define a periodic schedule as follows:

Define the time period T_{period} to be the least common multiple of the denominators of these rational values. Thus, in one time period, there will be an integral number of tasks sent over each link and executed by each node. We give each node sufficient buffer space to hold twice the number of tasks it receives per time period. Each task received in

period i will, in period $i+1$, either be computed locally or sent to a child. Since each node receives tasks from only one other node (its parent), there is no concern with scheduling the incoming communications to avoid conflicts. Further, each node is free to schedule its sends arbitrarily within a time period. Note that this schedule is substantially simpler than what is required when processors were connected as an arbitrary graph (cf., [19]).

A node at depth d does not receive any tasks during the first $d-1$ time periods and so will only enter the "steady-state mode" in time period d. Similarly, the root will eventually run out of tasks to send, so the final time periods will also be different from the steady state. It is often possible to improve the schedule in the start-up and clean-up time periods, which is the concern of the NP-complete makespan minimization problem. However, the periodic schedule described above is asymptotically optimal. More precisely, let z be the number of tasks executed by the periodic schedule in the steady state during d time periods, where d is the maximum depth of any node that executes a positive number of tasks. Then, our schedule will execute up to z fewer tasks than any possible (not necessarily periodic) schedule. More precisely, given a time bound B for the execution, it can be shown that the periodic schedule computes as many tasks of each type as the optimal up to a constant (independent of B) number of tasks. This result is an easy generalization of the same result with a single application [18], [19]. Note that as the applications we consider consist of a large number of independent tasks, z is generally much smaller than the total number of tasks of an application.

One final comment is that the time period T_{period} and the amount of buffer space used can be extraordinarily large, making this schedule impractical. We will revisit this issue later.

4.3 The Optimal Solution for Star Networks

When the computer platform is a star network, we can prove that the optimal solution has a very particular structure: applications with higher CCR are scheduled on processors with higher bandwidth. Thus, if we order the processors according to their bandwidths, then each application is executed by a set of consecutive nodes, which we refer to as a *slice*. The application with the highest CCR is executed by the first slice of processors, those with largest bandwidths. Then, the next most communication-intensive application is executed by the next slice of processors, and so on. There is a possible overlap at the slice boundaries. For instance, P_{a_1}, the processor at the boundary of the first two slices, may execute tasks for both applications A_1 and A_2.

To simplify matters, we consider the root P_0 to be a worker with infinite bandwidth ($b_0 = +\infty$). The following proposition proves that the optimal solution has the structure described above:

Proposition 1. *Sort nodes by bandwidth so that $b_0 \geq b_1 \ldots \geq b_p$ and sort the applications by CCR so that $\frac{b^{(1)}}{c^{(1)}} \geq \frac{b^{(2)}}{c^{(2)}} \ldots \geq \frac{b^{(K)}}{c^{(K)}}$. Then, there exist indices $a_0 \leq a_1 \ldots \leq a_K$ such that only processors P_u, $u \in [a_{k-1}, a_k]$, execute tasks of type k in the optimal solution.*

Proof. The key idea is to show that if a node P_i is assigned a task with a lower CCR than a task assigned to P_{i+1}, then these two nodes could swap an equal amount of computational work. This would reduce the communication time required by the schedule without changing any throughput. Thus, by a sequence of such swaps, any schedule can be transformed to one of the desired structure, without changing the fair throughput. See [2] for a detailed proof. □

This characterization does not enable the determination of the boundaries of the slices nor the $\alpha_u^{(k)}$ through analytical formulas. We did not succeed in deriving a counterpart of Proposition 1 for tree-shaped platforms. Intuitively, the problem is that a high-bandwidth child of node P_i can itself have a low-bandwidth high-computation-rate child, so there is no a priori reason to give P_i only communication-intensive tasks. Still, we use the intuition provided by Proposition 1 and its proof to design the heuristic in Section 5.5.

5 Demand-Driven and Decentralized Heuristics

As shown in Section 4.1, given a tree-shaped platform and the set of all application parameters, we are able to compute an optimal periodic schedule from any linear programming solution. However, this approach suffers from several serious drawbacks. First, the period of the schedule is the least common multiple of the denominators of the solution of the linear program (1). This period may be huge, requiring the nodes to have unreasonably large buffers to ensure uninterrupted steady-state behavior. The problem of buffer size has already been pointed out in [21] and [31], where it is shown that no finite amount of buffer space is sufficient for every tree. It is also known that finding the optimal throughput when buffer sizes are bounded is a strongly NP-hard problem even in very simple situations [31].

Since an unlimited buffer space is unrealistic, we will only consider *demand-driven* algorithms, which operate as follows: Each node has a local *worker* thread and a *scheduler* thread. The worker thread is an infinite loop that requests a task from the same node's scheduler thread and then, upon receiving a task, executes it. Fig. 3 shows the pseudocode for the scheduler thread. The "select" choices in line 5 depend on the particular heuristic used and can be based on, for instance, the history of requests, task types it has received, and the communication times it has observed for its children.

A second problem that some schedulers (including those generated as in Section 4.2) encounter is that centralized coordination becomes an issue as the size of the platform grows. It may be costly to collect up-to-date information and disseminate it to all nodes in the system. Consequently, a decentralized scheduling algorithm, where all choices are based exclusively on *locally* available information, is desirable.

In the following, we consider one demand-driven algorithm that is based on global information (derived from a solution to the linear programming problem) and four algorithms that are fully decentralized.

5.1 Centralized Linear-Programming-Based (*LP*)

If we know the computation power and communication speeds of all nodes in the distributed system, we can solve

```
1: Loop
2:    If there will be room in your buffer, request work from your parent.
3:    Get incoming requests from your local worker and children, if any.
4:    Move incoming tasks from your parent, if any, into your buffer.
5:    Select which thread (your local worker or a child's scheduler) to assign work to, and the type of
      application that will be assigned.
6:    If you have a request and a task that match your choice Then
7:       Send the task to the chosen thread (when the send port is free)
8:    Else
9:       Wait for a request or a task
```

Fig. 3. Demand-driven scheduler thread, ran in each node.

the linear programming problem (Section 4.1), obtaining values for the number of tasks of each type it should assign to each of its children during each time period. Thereafter, no further global communication is required.

Each scheduler thread uses a 1D load-balancing mechanism [32] to select a requesting thread and an application type. The 1D load-balancing mechanism works as follows: if task k should be chosen with frequency $f(k)$ and has already been chosen $g(k)$ times, then the next task to be sent will be of type ℓ, where $\frac{g(\ell)+1}{f(\ell)} = \min_k \frac{g(k)+1}{f(k)}$.

We might hope that the *LP* heuristic would always converge to the optimal throughput, but we will see in Section 6.2.1 that this is not always the case, primarily because of insufficient buffer space.

5.2 First-Come, First-Served (*FCFS*) Heuristic

The *FCFS* heuristic is a very simple and common decentralized heuristic. Each scheduler thread simply fulfills its requests on an *FCFS* basis, using the tasks it receives in order from its parent. The root ensures fairness by selecting task types using a round-robin selection. This simple heuristic has the disadvantage, not shared by the other methods we consider, that an extremely slow communication link cannot be avoided. Thus, optimal performance should not be expected.

5.3 Coarse-Grain Bandwidth-Centric (*CGBC*) Heuristic

This heuristic (*CGBC*) builds upon our previous work for scheduling a single application on a tree-shaped platform [18], [19]. In bandwidth-centric scheduling, each node only needs to know the bandwidth to each of its children. The node's own worker thread is considered to be a child with infinite bandwidth. The scheduler thread prioritizes its children in order of bandwidth, so the greatest bandwidth has the highest priority. The scheduler always assigns tasks to the highest priority requester. Bandwidth-centric scheduling has been shown to have an optimal steady-state throughput for a single application, both theoretically and, when the buffers are sufficiently large, in extensive simulations.

Our *coarse-grain* heuristic assembles several tasks into a large one. More precisely, we build a macrotask out of one task of type k for each k, and the macrotasks are scheduled using the bandwidth-centric method. Thus, fairness is assured.

Unfortunately, even though bandwidth-centric scheduling can give the optimal throughput of macrotasks, the *CGBC* heuristic does not reach the optimal fair throughput. Indeed, Proposition 1 asserts that in star networks, nodes with faster incoming links should process only tasks with larger CCRs. However, since a macrotask includes tasks of *all* types, *CGBC* will send communication-intensive tasks to some low-bandwidth nodes in a star network.

5.4 Parallel Bandwidth-Centric (*BCS*) Heuristic

The *BCS* heuristic superposes bandwidth-centric trees for each type of task, running all of them in parallel. More precisely, each node has K scheduler and K worker threads that run concurrently, corresponding to the K application types. Threads only communicate with other threads of their own type.

In all our other simulations, we enforce the one-port constraint for each scheduler thread. However, for this *BCS* heuristic, we have not enforced this constraint globally across the schedulers, and a node may send as many as K tasks concurrently, one of each type. Instead, we model the contention on the port, so the aggregate bandwidth does not exceed the port's limit (similarly, the node's processor can multitask between multiple tasks). This gives the *BCS* strategy an unfair advantage over the other heuristics. In fact, it has been shown [21] that allowing *interruptible communication* (which is similar to concurrent communication) dramatically reduces the amount of buffer space needed to achieve optimal throughput.

5.5 Data-Centric Scheduling (*DATA-CENTRIC*) Heuristic

This heuristic is our best attempt to design a decentralized demand-driven algorithm that converges to a solution of the linear program (1). The idea is to start from the bandwidth-centric solution for the most communication-intensive application and to progressively trade some of these tasks for more computationally intensive ones. Doing so yields better values for the expected $\alpha_u^{(k)}$'s and the expected $sent_{u\to v}^{(k)}$'s, which can in turn be used in the demand-driven algorithm in Fig. 3. These frequencies are continuously recomputed so as to cope with potential availability variations. The rest of this section is devoted to the details of the *trading* operations. As we have explained in Section 3.4, in the optimal solution on trees, all applications have the same throughput. Therefore, this heuristic starts from an initial solution and updates the $\alpha_u^{(k)}$'s until all throughputs are close to each other.

We sort the task types by nonincreasing CCRs. We start the algorithm using the pure bandwidth-centric approach for tasks of type 1, but as the computation proceeds, a node will find itself receiving a mix of different types of tasks. To

reduce the imbalance, the root iteratively applies the four operations described below, in the listed order of precedence. In the following, H (respectively, L) denotes the application that currently has the highest (respectively, lowest) throughput. As the root distributes all tasks, H and L are easy to identify. Those operations attempt to increase the number of tasks of type L that are assigned, sometimes by reducing the number of H's.

Communication trading. Suppose H has a higher CCR than L (which is the common case since we start with only tasks of type 1). Then, if a child reports that it is not fully utilized (either because its CPU is idle or because it cannot keep up with the requests it receives from its own children, that is, $\sum_k \alpha_u^{(k)} c^{(k)} < c_u$, or $\sum_{v\in\Gamma(u)} \frac{\sum_k sent_{u\to v}^{(k)} \cdot b^{(k)}}{b_v} = 1$), then the parent can *substitute* some tasks of type H by sending them in place of some tasks of type L to the underutilized child. It should make the substitution in a way that keeps the communication time the same (that is, in the ratio of $b^{(L)}$ H's for $b^{(H)}$ L's) and limited by the number that would make the throughputs equal. Last, let CPU denote the CPU computation time to execute all tasks currently assigned to processor P_u. Then, we have $CPU = \sum_k \frac{\alpha_u^{(k)} . c^{(k)}}{c_u}$, and we should not exceed the CPU capacity after the update. Therefore, $\alpha_u^{(L)}$ is increased by ε_L, and $\alpha_u^{(H)}$ is decreased by ε_H with the following constraints:

$$\begin{cases} \varepsilon_L b^{(L)} = \varepsilon_H b^{(H)} \\ 0 \le \alpha^L + \varepsilon_L \le \alpha^H - \varepsilon_H \\ CPU + \varepsilon_L \frac{c^{(L)}}{c_u} - \varepsilon_H \frac{c^{(H)}}{c_u} \le 1. \end{cases}$$

Hence, we get

$$\varepsilon_H = \min\left(\frac{\alpha^{(H)} - \alpha^{(L)}}{1 + \frac{b^{(H)}}{b^{(L)}}}, \frac{1 - CPU}{\frac{c^{(H)} c^{(L)}}{b^{(L)} c_u} \left(\frac{b^{(H)}}{c^{(H)}} - \frac{b^{(L)}}{c^{(L)}} \right)} \right).$$

Gap filling. Suppose that some bandwidth from the root is not used and that a remote processor P_u could receive more tasks of a low-throughput application. This step calculates ε_L, the possible additional number of tasks of type L that this processor could handle. Let CPU denote the CPU computation time to execute all tasks currently assigned to processor P_u. Then, we have $CPU = \sum_k \frac{\alpha_u^{(k)} . c^{(k)}}{c_u}$, and the following computation limit on ε_L has to hold: $CPU + \varepsilon_L \frac{c^{(L)}}{c_u} \le 1$. In addition, there must be enough free bandwidth along the path from the root node to P_u to send the additional tasks; therefore, for any node i along this path, we can define $bus\_occupation(p(i)) = \sum_k \sum_j \frac{sent_{p(i)\to j}^{(k)} . b^{(k)}}{b_j}$, and we need the following condition on ε_L to hold true: $bus\_occupation(p(i)) + \varepsilon_L \frac{b^{(L)}}{b_i} \le 1$ (Fig. 4a).

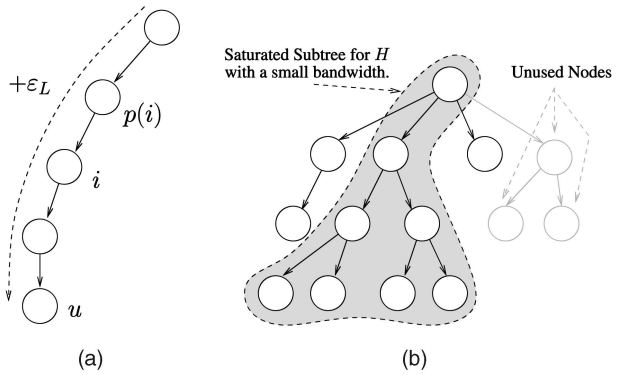

Fig. 4. Rebalancing operations. (a) Gap filling. (b) Bus desaturation.

Last, to avoid overreducing the imbalance between $\alpha^{(H)}$ and $\alpha^{(L)}$, we add the following constraint: $\alpha^{(H)} \ge \alpha^{(L)} + \varepsilon_L$. Therefore, we have

$$\varepsilon_L = \min\left(\frac{c_u(1 - CPU)}{c^{(L)}}, \alpha^{(H)} - \alpha^{(L)}, \min_{\substack{i \in \text{path from} \\ \text{the root to } P_u}} \left(\frac{b_i(1 - bus\_occupation(p(i)))}{b^{(L)}} \right) \right).$$

Bus desaturation. If the bus is saturated by tasks with a high CCR, we may still be using only workers with high communication capacity. However, the workload might still be increased by using additional idle subtrees, that is, the current tree must be *widened*. Thus, we need to reduce the amount of tasks of type H that are processed by the currently used subtrees. This heuristic takes the branch with the smallest bandwidth that processes tasks of type H and scales down the $\alpha_i^{(H)}$ and $sent_{i\to j}^{(H)}$ values of nodes on the branch by 10 percent (Fig. 4b). This operation allows us to decrease the communication resource utilization and precedes the next round of "*Gap filling*" operations.

Task trading on the root. At some point, it may be the case that application H is processed only on the root node. This heuristic will try to substitute ε_H tasks of type H for ε_L tasks of type L at the root. To do so, we need the following constraints: $\varepsilon_H \le \alpha_{root}^{(H)}$, $\alpha^{(H)} - \varepsilon_H \ge \alpha^{(L)} + \varepsilon_L$, and $\varepsilon_L . \frac{c^{(L)}}{c_{root}} = \varepsilon_H . \frac{c^{(H)}}{c_{root}}$. Therefore, we have

$$\varepsilon_H = \min\left(\alpha_{root}^{(k)}, \frac{\alpha^{(H)} - \alpha^{(L)}}{1 + \frac{c^{(H)}}{c^{(L)}}} \right) \text{ and } \varepsilon_L = \frac{c^{(H)}}{c^{(L)}} \varepsilon_H.$$

The preceding operations are iteratively performed (with the listed order of precedence) until we reach a (tunable) balance, for example

$$\frac{\max_k \{\alpha^{(k)}\} - \min_k \{\alpha^{(k)}\}}{\min_k \{\alpha^{(k)}\}} < 0.05.$$

The above operations may appear to need global knowledge about the tree. For example, it may seem at first sight that when performing a "*Gap filling*" operation, the master needs to know the path to its remote descendant P_u. However, this operation in fact simply amounts to computing a minimum along this path, which can be done via a classical and efficient distributed propagation mechanism. The same distributed technique can be used for all other

operations as they only require information from immediate descendants in a single subtree.

6 Simulation Results

6.1 Evaluation Methodology

6.1.1 Throughput Evaluation

It is not at all obvious how to determine that a computation has entered the steady state, and measuring throughput becomes even trickier when the schedule is not periodic. We took a pragmatic heuristic approach for our experiments. Let T denote the earliest time that all tasks of some application were completed. Let $N^{(k)}(t)$ denote the number of tasks of type k that were finished in time period $[0, t]$. We define the achieved throughput ρ_k for application k by

$$\rho_k = \frac{N^{(k)}((1-\varepsilon)T) - N^{(k)}(\varepsilon T)}{(1-2\varepsilon)T}, \text{ where } \varepsilon \in [0, 0.5].$$

The ε factor allows us to ignore the initial and final instabilities (in practice, we set ε to be equal to 0.1). In the following, we will refer to ρ_k as the *experimental throughput* of application k as opposed to the expected throughput that can be computed by solving the linear program (1). Likewise, the minimum of the experimental throughputs is called the *experimental fair throughput*.

6.1.2 Platform Generation

The platforms used in our experiments are random trees described by two parameters: the number of nodes n and the maximum degree $degree_{max}$. To generate the interconnection network topology, we use a breadth-first algorithm (see [2] for more details) in order to have wide (rather than deep and narrow) trees. In our experiments, we generated a total of 150 trees, 10 trees each with 5, 10, 20, 50, and 100 nodes and a maximum degree of 2, 5, or 15.

We assigned capacity, latency, and CPU floating-point rate values on edges and nodes at random. Those values come from real measurements (performed using tools like pathchar) on machines spread across the Internet. CPU rates ranged from 22.151 Mflops (an old Pentium Pro 200-MHz processor) to 171.667 Mflops (an Athlon 1800). The bandwidth ranged from 110 Kbps to 7 Mbps, and latency ranged from 6 ms to 10 sec. Note that in the SimGrid simulator [3] that we are using, latency and link capacity are limiting factors for determining the effective bandwidth of a connection.

6.1.3 Application Generation

An application is mainly characterized by its *CCR*. We used $CCR_{min} = 0.001$, which corresponds to the computationally intensive task of multiplying two 3,500 × 3,500 matrices. We also set an upper bound for *CCR* of 4.6, corresponding to the addition of two such matrices. In choosing application types, we picked CCR_{max} between 0.002 and 4.6 and then chose the applications' *CCR*s to be evenly spaced in the range $[CCR_{min}, CCR_{max}]$. For simplicity, we take K to be 2 or 3.

6.1.4 Heuristic Implementation

The experiments were performed using the SimGrid simulator [3], and the values of c_i and b_i were measured from within the simulator (based on the node values input) and used to make the decisions in the algorithms in Section 5.

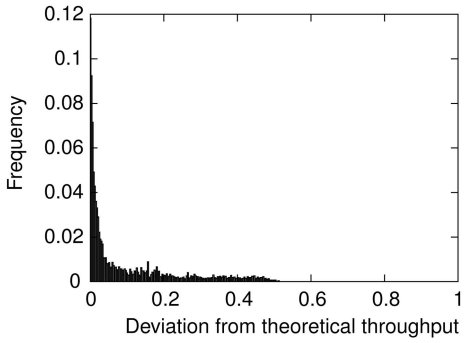

Fig. 5. Deviation of experimental fair throughput from expected theoretical throughput $(1 - \frac{\text{Experimental fair throughput}}{\text{Theoretical fair throughput}})$ for *LP*, *DATA-CENTRIC*, and *CGBC*.

As explained in Section 5.5, the demand-driven algorithms send requests (involving a few bytes) from children to parents. Our simulations included the request mechanism, and we ensured that no deadlock occurred in our thousands of experiments, even when some load variations occurred. Except where otherwise noted, throughput evaluations were performed using 200 tasks per application. Note that we carefully checked by hand using a larger number of tasks (10 times more tasks) that the smaller number we used for our experiments was always sufficient to reach the steady state (the experimental throughput was within 1 percent for all configurations when adding more tasks).

6.2 Case Study

6.2.1 Theoretical versus Observed Throughput

For the heuristics *LP*, *DATA-CENTRIC*, and *CGBC*, we can easily compute the expected *theoretical* fair throughput as they all rely on explicit rate computation. This allowed us to explore how implementation issues result in the experimentally obtained fair throughput differing from the theoretical fair throughput. There are many reasons why these quantities might differ, such as the overhead of the request mechanism or a start-up period longer than the 10 percent we allowed for. It turned out that the major cause of inefficiency was the limit on the buffer size.

Our experiments assumed enough buffer space to hold 10 tasks of any type. For this case, Fig. 5 depicts the experimental fair throughput deviation from the expected theoretical throughput for heuristics *CGBC*, *LP*, and *DATA-CENTRIC*. This deviation is computed as $1 - \frac{\text{Experimental fair throughput}}{\text{Theoretical fair throughput}}$. All three heuristics exhibited a similar distribution, so they were combined in this figure. The average deviation is equal to 9.426 percent. However, when we increased the buffer size by a factor 10 (and increased the number of tasks per application to 2,000), the mean average deviation dropped to 0.334 percent. Even though the larger buffer size led to a much better throughput, we considered it unrealistic and used size 10 in all other experiments.

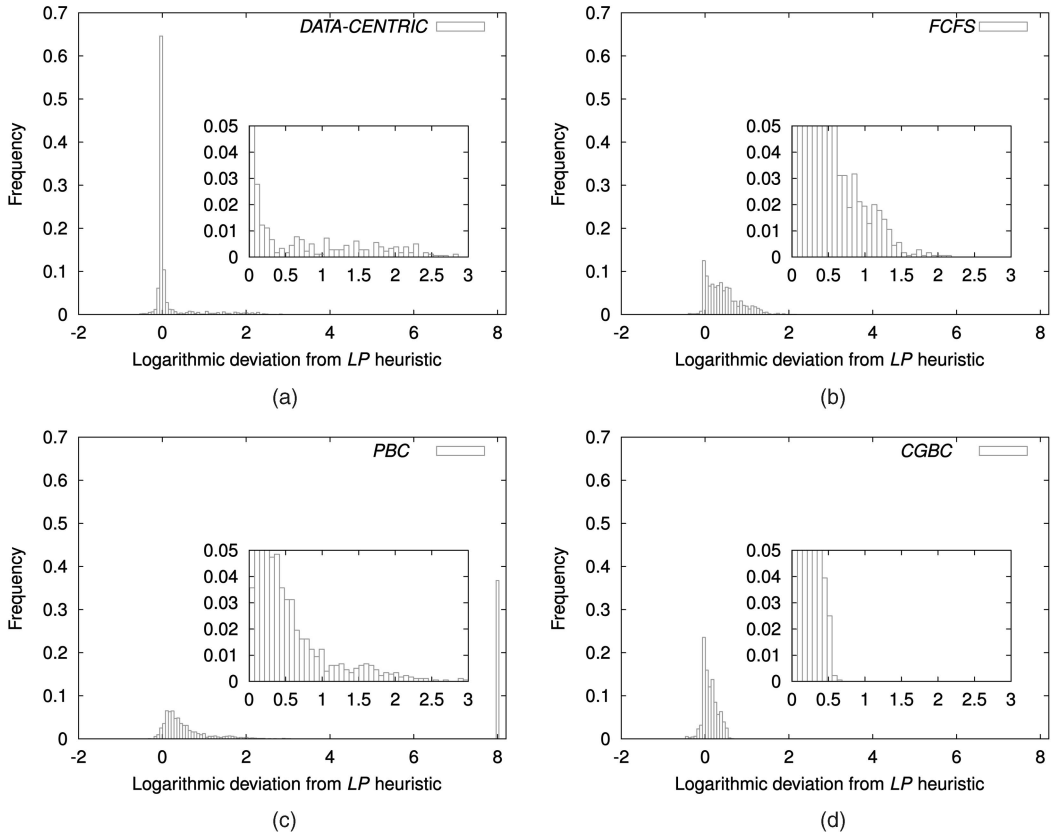

Fig. 6. Logarithmic deviation from *LP* performances $\left(\ln\left(\frac{\min_k \alpha^{(k)} \text{ for } LP}{\min_k \alpha^{(k)} \text{ for other heuristic}}\right)\right)$. (a) Performances of *DATA-CENTRIC*. (b) Performances of *FCFS*. (c) Performances of *BCS*. (d) Performances of *CGBC*.

6.2.2 Performance of Heuristics

Let us first compare the relative performances of our five heuristics (*FCFS*, *BCS*, *CGBC*, *LP*, and *DATA-CENTRIC*). More precisely, for each experimental setting (that is, a given platform and a given *CCR* interval), we compute the (natural) logarithm of the ratio of the experimental fair throughput of *LP* with the experimental fair throughput of a given heuristic (applying a logarithm enables us to have a symmetrical value). This value is called in the following *logarithmic deviation*. Therefore, a positive value means that *LP* performed better than the other heuristic. Fig. 6 depicts the histogram plots of these values.

First of all, we observe that most values are positive, which illustrates the superiority of *LP*. Next, observe in Fig. 6a that *DATA-CENTRIC* is very close to *LP* most of the time, despite the distributed computation of the expected $\alpha_u^{(k)}$ and $sent_{u\to v}^{(k)}$ values. However, the geometric average[2] of these ratios is equal to 1.164, which is slightly larger than the geometric average for *CGBC* (1.156). The reason is that even though in most settings *DATA-CENTRIC* ends up with a very good solution, in a few instances, it performed very badly (up to 16 times worse than *LP*). In contrast, *CGBC* (see Fig. 6d) is much more stable since its worst performance is only twice that of *LP*. Note that these failures happen on any type of tree (small or large, narrow or wide) and that the geometric averages of these two heuristics are always very close to each other. We have also checked that these failures are not due to an artifact of the decentralized control of the scheduling by ensuring that the theoretical throughput has the same behavior (that is, the bad behavior actually comes from the computation of the expected $\alpha_u^{(k)}$ and $sent_{u\to v}^{(k)}$). We are still investigating the reasons why *DATA-CENTRIC* fails on some instances and suspect that it is due to the use of the (sometimes misleading) intuition of Proposition 1. Indeed, in this heuristic, applications with a high CCR are performed mainly on the subtrees with the best bandwidths at the root, whereas applications with a low CCR are performed primarily on the subtrees with the worst bandwidths at the root, which is definitely not optimal on particular instances.

Unsurprisingly, *BCS* leads to very bad results. In many situations (more than 35 percent), an application has been particularly unfavored, and the fair experimental throughput was close to 0. The logarithm of the deviation for these situations has been normalized to 8. These poor results advocate the need for fairness guarantees in distributed computing environments like the ones we consider. As a matter of fact, this is somehow similar to the *Bushel of AppLeS* problem [34]: all applications identify the same resources as "best" for their tasks and seek to use them at the same time, thus making them no longer optimal or available.

Last, the geometrical average of *FCFS* is 1.564, and in the worst case, its performance is more than eight times worse than *LP*. On the average, *FCFS* is therefore much worse than

2. It is a well-known fact [33] that the arithmetic average of ratios can lead to contradictory conclusions when changing the reference point. Therefore, we use a geometric average of ratios, which is known to be closer to the common idea of *average ratio*.

LP. On small platforms, the performances for *FCFS* and *CGBC* have the same order of magnitude. However, on larger ones (50 and 100 workers), *CGBC* performs much better (geometrical average equal to 1.243) than *FCFS* (geometrical average equal to 2.0399).

7 Conclusion

In this paper, we presented several heuristics for scheduling multiple Bag-of-Tasks applications on a tree-connected platform composed of heterogeneous processing and communication resources. Our contributions to this problem are the following:

- We outlined a theoretical scheduling model for Bag-of-Tasks applications' scheduling environments with a particular emphasis on the fairness issue.
- We presented a centralized algorithm that, given the performance of all resources, computes an optimal schedule with respect to throughput maximization for tree networks. We also have characterized an optimal solution on single-level trees.
- Since centralized algorithms may be unrealistic on large-scale platforms, we then presented several distributed algorithms based on decentralized heuristics.
- We evaluated the efficacy of these heuristics using a wide range of realistic simulation scenarios, using a limited buffer space. The results obtained by the most sophisticated heuristics are quite reasonable compared to the optimal centralized algorithm.

The experimental analysis provided in this article is an average-case analysis on small data sets. It would be certainly be instructive to perform a worst-case analysis for the previous heuristics. In particular, we conjecture that it should be possible to establish a performance guarantee (an approximation factor) for *CGBC* and *FCFS*. However, we believe that *DATA-CENTRIC* and *BCS* are much harder to analyze.

We have seen with the *BCS* heuristic that noncooperative approaches, where each application optimizes its own throughput, lead to a particularly unfair Nash equilibrium [35]. This Nash equilibrium has been recently analytically studied in a multiport context [36]. Extending this work to a single-port setting seems very challenging.

Another approach could be a cooperative approach where several decision-makers (each of them being responsible for a given application) cooperate in making the decisions such that each of them will operate at its optimum. This situation can be modeled as a cooperative game like in [37], where this approach is successfully used for simpler problems such as bandwidth sharing in networks and other fairness criteria (proportional fairness). In particular, fully distributed protocols achieving proportionally fair allocations have been proposed. However in our situation, hierarchical resource sharing is rather hard to model, which renders such an approach quite challenging.

Acknowledgments

The authors would like to thank the reviewers for their detailed comments and suggestions, which greatly helped us to improve the paper.

References

[1] *Berkeley Open Infrastructure for Network Computing,* http://boinc.berkeley.edu, 2007.
[2] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert, "Scheduling Multiple Bags of Tasks on Heterogeneous Master-Worker Platforms: Centralized versus Distributed Solutions," Technical Report 2005-45, LIP, Sept. 2005.
[3] A. Legrand, L. Marchal, and H. Casanova, "Scheduling Distributed Applications: The SIMGRID Simulation Framework," *Proc. Third IEEE Int'l Symp. Cluster Computing and the Grid (CCGrid '03),* May 2003.
[4] SETI, http://setiathome.ssl.berkeley.edu, 2007.
[5] J. Cowie, B. Dodson, R.-M. Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery, and J. Zayer, "A World Wide Number Field Sieve Factoring Record: On to 512 Bits," *Advances in Cryptology—Proc. Int'l Conf. Theory and Applications of Cryptology and Information Security (Asiacrypt '96),* pp. 382-394, 1996.
[6] Prime, http://www.mersenne.org, 2007.
[7] Entropia, http://www.entropia.com, 2007.
[8] M. Litzkow, M. Livny, and M.W. Mutka, "Condor—A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. Distributed Computing Systems (ICDCS '88),* pp. 104-111, 1988.
[9] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive Computing on the Grid Using AppLeS," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 4, pp. 369-382, Apr. 2003.
[10] H. Casanova and F. Berman, "Parameter Sweeps on the Grid with APST," *Grid Computing,* F. Berman, G. Fox, and T. Hey, eds. John Wiley & Sons, 2002.
[11] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Proc. 10th IEEE Symp. High Performance Distributed Computing (HPDC-10 '01),* Aug. 2001.
[12] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F.A.B. da Silva, C.O. Barros, and C. Silveira, "Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach," *Proc. 32nd Int'l Conf. Parallel Processing (ICCP '03),* Oct. 2003.
[13] M.R. Garey and D.S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.
[14] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE CS Press, 1995.
[15] *Scheduling Theory and its Applications,* P. Chrétienne, E. G. Coffman Jr., J.K. Lenstra, and Z. Liu, eds. John Wiley & Sons, 1995.
[16] D. Bertsimas and D. Gamarnik, "Asymptotically Optimal Algorithm for Job Shop Scheduling and Packet Routing," *J. Algorithms,* vol. 33, no. 2, pp. 296-318, 1999.
[17] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Steady-State Scheduling on Heterogeneous Clusters," *Int'l J. Foundations of Computer Science,* vol. 16, no. 2, pp. 163-194, 2005.
[18] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2002.
[19] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, "Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms," *IEEE Trans. Parallel and Distributed Systems,* vol. 15, no. 4, pp. 319-330, Apr. 2004.
[20] B. Kreaseck, "Dynamic Autonomous Scheduling on Heterogeneous Systems," PhD dissertation, Univ. of California, San Diego, 2003.
[21] L. Carter, H. Casanova, J. Ferrante, and B. Kreaseck, "Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2003.
[22] B. Hong and V. Prasanna, "Distributed Adaptive Task Allocation in Heterogeneous Computing Environments to Maximize Throughput," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2004.
[23] M. Mowbrayy, F. Brasileiroz, N. Andradez, J. Santanaz, and W. Cirnez, "Reciprocation-Based Economy for Multiple Services in Peer-to-Peer Grids," *Proc. Sixth IEEE Int'l Conf. Peer-to-Peer Computing (P2P '06),* Sept. 2006.
[24] T. Bonald and L. Massoulié, "Impact of Fairness on Internet Performance," *Proc. ACM SIGMETRICS/Performance '01,* pp. 82-91, 2001.

[25] L. Massoulié and J. Roberts, "Bandwidth Sharing: Objectives and Algorithms," *IEEE/ACM Trans. Networking*, vol. 10, no. 3, pp. 320-328, June 2002.
[26] P. Bhat, C. Raghavendra, and V. Prasanna, "Efficient Collective Communication in Distributed Heterogeneous Systems," *Proc. 19th Int'l Conf. Distributed Computing Systems (ICDCS '99)*, pp. 15-24, 1999.
[27] P. Bhat, C. Raghavendra, and V. Prasanna, "Efficient Collective Communication in Distributed Heterogeneous Systems," *J. Parallel and Distributed Computing*, vol. 63, pp. 251-263, 2003.
[28] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Pipelining Broadcasts on Heterogeneous Platforms," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 4, pp. 300-313, Apr. 2005.
[29] O. Beaumont and L. Marchal, "Pipelining Broadcasts on Heterogeneous Platforms under the One-Port Model," Research Report RR-2004-36, LIP, ENS Lyon, France, July 2004.
[30] D. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall, 1987.
[31] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Independent and Divisible Tasks Scheduling on Heterogeneous Star-Shaped Platforms with Limited Memory," *Proc. 13th Euromicro Workshop Parallel, Distributed and Network-Based Processing (PDP '05)*, pp. 179-186, 2005.
[32] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)," *IEEE Trans. Computers*, vol. 50, no. 10, pp. 1052-1070, Oct. 2001.
[33] R. Jay, *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.
[34] F. Berman and R. Wolski, "TheAppLeS Project: A Status Report," *Proc. Eighth NEC Research Symp.*, http://www.gcl.ucsd.edu//hetpubs.html#AppLeS, 1997.
[35] J.F. Nash, "Equilibrium Points in N-Person Games," *Proc. Nat'l Academy of Sciences USA*, vol. 36, pp. 48-49, 1950.
[36] A. Legrand and C. Touati, "Non-Cooperative Scheduling of Multiple Bag-of-Task Applications," *Proc. IEEE INFOCOM '07*, 2007.
[37] H. Yaïche, R.R. Mazumdar, and C. Rosenberg, "A Game Theoretic Framework for Bandwidth Allocation and Pricing in Broadband Networks," *IEEE/ACM Trans. Networking*, vol. 8, no. 5, pp. 667-678, 2000.

Olivier Beaumont received the PhD degree from the Université de Rennes in January 1999. He is currently an associate professor in the Computer Science Laboratory LaBRI, 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, Berkeley, in 1974. He worked as a research staff member and manager at IBM T.J. Watson Research Center for nearly 20 years in the areas of probabilistic algorithms, compilers, VLSI testing, and high-performance computation. Since 1994, he has been a professor in the Computer Science and Engineering Department, University of California, San Diego, where he served as the vice chair and then the chair of the department between 1996 and 2000. His current research interests include scientific computation, performance programming, parallel computation, and computer architecture. He 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 the Massachusetts Institute of Technology (MIT) in 1974. She was a research staff member at the IBM T.J. Watson Research Center from 1978 to 1994. She is currently 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 received the PhD degree from the Ecole Normale Supérieure de Lyon in 2003. He is currently a CNRS researcher in the Computer Science Laboratory LIG, Grenoble. He is mainly interested in parallel algorithm design for heterogeneous platforms and in scheduling techniques.

Loris Marchal received the PhD degree from Ecole Normale Supérieure de Lyon (ENS Lyon) in 2006. He is currently a CNRS researcher in the Computer Science Laboratory LIP, Ecole Normale Supérieure de Lyon. He is mainly interested in communication optimization on heterogeneous platforms and in scheduling techniques.

Yves Robert received the PhD degree from the Institut National Polytechnique de Grenoble in January 1986. He is currently a full professor in the Computer Science Laboratory LIP, Ecole Normale Supérieure de Lyon. He is the author of four books, 80 papers published in international journals, and 100 papers published in international conferences. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. His main research interests are scheduling techniques and parallel algorithms for clusters and grids. He is a member of the ACM and the IEEE.