Centralized versus distributed schedulers for multiple bag-of-task applications

Olivier Beaumont[1], Larry Carter[2], Jeanne Ferrante[2],
Arnaud Legrand[3], Loris Marchal[4] and Yves Robert[4]

[1]Laboratoire LaBRI,
CNRS-INRIA Bordeaux, France
Olivier.Beaumont@labri.fr

[2]Dept. of Computer Science and Engineering,
University of California, San Diego, USA
{carter,ferrante}@cs.ucsd.edu

[3]Laboratoire ID-IMAG
CNRS-INRIA,Grenoble, France
Arnaud.Legrand@imag.fr

[4]Laboratoire LIP
CNRS-INRIA,École Normale Supérieure de Lyon, France
{Loris.Marchal,Yves.Robert}@ens-lyon.fr

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 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. Each application is given a weight that quantifies its relative value. The goal of scheduling is to maximize throughput while executing tasks from each application in the same ratio as their weights.

We can find the optimal asymptotic rates by solving a linear program that expresses all necessary problem constraints, and we show how to construct a periodic schedule. For single-level trees, the solution is characterized by processing tasks with larger communication-to-computation ratios at children with larger bandwidths. For multi-level 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 a centralized solution obtained via linear programming. The best of our decentralized heuristics achieves the same performance on about two-thirds of our test cases, but is far worse in a few cases. While our results are based on simplistic assumptions and do not explore all parameters (such as buffer size), they provide insight into the important question of fairly and optimally co-scheduling heterogeneous applications on heterogeneous grids.*

1. Introduction

In this paper, we consider the problem of scheduling multiple applications that are executed concurrently, 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 multi-level 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 equal-sized tasks, and all tasks originate at the tree's root. The applications can be very different in nature, e.g. 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* for all of its tasks. This ratio proves to be an important parameter in the scheduling process. This scenario is somewhat similar to that addressed by existing systems. For instance BOINC [10] is a centralized scheduler that distributes tasks for participating applications, such as SETI@home, ClimatePrediction.NET, and Einstein@Home.

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 throughput. For each application, the root node must decide which workers (i.e. which subtree) the tasks are sent to. For multi-level trees, each non-leaf 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 are equally important, the scheduler should aim to process the same number of tasks for each application. We 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.

For each application A_k, let $\nu^{(k)}(t)$ be the number of tasks of A_k completed by time t. At any given time t, we can define the throughput $\alpha^{(k)}$ of application k to be $\nu^{(k)}(t)/t$. To balance the tasks according to the specified priority weights, we use the objective function MAXIMIZE $\min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\}$. This function, called *fair throughput* in the following, corresponds to the well-known MAX-MIN fairness strategy [7, 17] among the different applications, with coefficients $1/w^{(k)}$.

We will consider both centralized and decentralized schedulers. For smaller platforms it may be realistic to 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, i.e. 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 higher throughput. For single-level rooted trees, we provide an interesting characterization of the optimal solution: applications with larger communication-to-computation ratio should be processed by the workers with larger bandwidths, independent of the communication-to-computation ratios 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 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 communication-to-computation ratios. We evaluate the decentralized heuristics through extensive simulations using SimGrid [16], and use a centralized algorithm (guided by the linear program solution) as a reference basis.

The rest of the paper is organized as follows. In Section 2, we state precisely the scheduling problem under consideration, with all application and platform parameters, and the objective function used. Section 3 explains how to analytically compute the best solution, using a linear programming approach, and characterizes the solution for single-level trees. Then Section 4 deals with the design of several decentralized scheduling heuristics, while Section 5 provides an experimental comparison of these heuristics. Finally, we state some concluding remarks in Section 6.[1]

2. Platform and Application Model

In this paper, we make a number of overly simplistic assumptions; nevertheless, we believe that both by the theory and the experiments provide insight into the important question of how to optimally and fairly co-schedule heterogeneous applications on heterogeneous grids.

2.1. 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", $P_1, P_2, \dots, P_p$; 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. We ignore processor-task affinities; instead, we assume one can express the computational requirements of tasks as a number of floating-point operations, and that processor P_u can execute c_u floating-point operations per second (independent of which application it is executing).

There are several scenarios for the operation of the processors, which are discussed in Section A3 of the Appendix. In this paper, we concentrate on the *full overlap, single-port model* [8, 9]. 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.

1 Due to lack of space, we do not provide related work in this article but a thorough survey can be found in the extended version of this article [**?**].

2.2. Application Model

We consider K applications, A_k, $1 \leqslant k \leqslant K$. The root node P_0 initially holds all the input data necessary for each application A_k. Each application has a *priority weight* $w^{(k)}$ as described earlier. Each application is composed of a set of independent, same-size tasks. We can think of each A_k as 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 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 outwards from the root, i.e. 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. The *communication-to-computation ratio* of tasks of type k is defined as $b^{(k)}/c^{(k)}$.

Note that our notations use indices for platform resources (bandwidth b_u, CPU speed c_u) and exponents for application parameters (bytes $b^{(k)}$, floating-point operations $c^{(k)}$, weight $w^{(k)}$).

.

2.3. Objective Function

If each application had an unlimited supply of tasks, our objective function would be

$$\textsc{Maximize} \lim_{t\to\infty} \min_k \left\{ \frac{\nu^{(k)}(t)}{w^{(k)} \cdot t} \right\} \tag{1}$$

where $\nu^{(k)}(t)$ is the number of tasks of application A_k completed by time t. However, we can do better than studying asymptotic behavior. Following standard practice, we optimize the "steady-state throughput", i.e.

$$\textsc{Maximize} \quad \min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\}. \tag{2}$$

where $\alpha^{(k)}$ is the average number of tasks of A_k executed per time unit. There are two features of this approach. First, if we can derive an upper bound on the steady-state throughput for arbitrarily long periods, then this is an upper bound on the limit of formula (1). Second, if we construct a *periodic schedule* – one that begins and ends in exactly the same state – then the periodic schedule's throughput will be a lower bound on the limit of formula (1). Thus, this approach allows us to derive optimal results. When the number of tasks per application is large, we believe the advantage of avoiding the NP-completeness of the makespan optimization problem outweighs the disadvantage of not knowing the exact length of the start-up and clean-up phases of a finite schedule.

3. Computing the Optimal Solution

In this section, we show how to compute 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 4.

3.1. Linear Programming Solution

A summary of our notation 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 $u \neq 0$, can receive b_u bytes from its parent $P_{p(u)}$.
- Application k has weight $w^{(k)}$, and each task of type k involves $b^{(k)}$ bytes and $c^{(k)}$ floating-point operations.

We use linear programming to solve for the variables:

- $\alpha_u^{(k)}$, the number of tasks of type k executed by P_u each time unit.
- $\alpha^{(k)}$, the total number of tasks of type k executed per time unit.
- $sent_{u\to v}^{(k)}$, the number of tasks of type k received by P_v from $P_{p(v)}$ per time unit.

Any feasible schedule must be a solution to the linear programming problem:

$$\textsc{Maximize} \;\; \min_k \left\{ \frac{\alpha^{(k)}}{w^{(k)}} \right\} \text{ UNDER THE CONSTRAINTS}$$
$$\left\{ \begin{array}{ll} \forall k, \quad \sum_u \alpha_u^{(k)} = \alpha^{(k)} & \text{(definition of } \alpha^{(k)}\text{)} \\ \forall k, \forall u \neq 0, \quad sent_{p(u)\to u}^{(k)} = \alpha_u^{(k)} + \sum_{v\in\Gamma(u)} sent_{u\to v}^{(k)} & \\ & \text{(data movement conservation)} \\ \forall u, \quad \sum_k \alpha_u^{(k)} \cdot c^{(k)} \leqslant c_u & \\ & \text{(computation limit at node } P_u\text{)} \\ \forall u, \quad \sum_{v\in\Gamma(u)} \frac{\sum_k sent_{u\to v}^{(k)} \cdot b^{(k)}}{b_v} \leqslant 1 & \\ & \text{(communication limit out of } P_u\text{)} \\ \forall k, u \quad \alpha_u^{(k)} \geqslant 0 \text{ and } \quad sent_{u\to v}^{(k)} \geqslant 0 & \\ & \text{(non-negativity)} \end{array} \right. \tag{3}$$

We assume that all the parameters to the linear programming problem are rational numbers, and hence the solution will be rational also.

3.2. Reconstructing a Periodic Schedule

Suppose we have solved linear program (3). The conditions in the linear program deal with steady state behavior, but it may not be immediately obvious that there exists a valid schedule, where precedence constraints are satisfied (i.e. a task is processed on a processor only when the corresponding input file has been routed to this processor), that achieves the desired throughput. Nevertheless, suppose we have determined all the values $\alpha_u^{(k)}$, and $sent_{u\to v}^{(k)}$. Define the time period T_{period} to be the least common multiple of the denominators of these rational numbers. 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 it receives 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 receives to avoid conflicts. Further, each node is free to schedule its sends arbitrarily within a time period. Thus, this schedule is substantially easier than if the processors were connected as an arbitrary graph (c.f. [3]).

A node at depth d doesn't receive any tasks during the first $d-1$ time periods, so will only enter "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. 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 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 at most z fewer tasks than any possible (not necessarily periodic) schedule.

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.

3.3. The Optimal Solution for Star Networks

When the computer platform is a star network, we can prove the optimal solution has a very particular structure: Informally, each application is executed by a slice of consecutive nodes. The application with the highest communication-to-computation ratio is executed by a 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 between the slices. 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 notations in the following proposition, we consider the root P_0 as a worker with infinite bandwidth ($b_0 = +\infty$):

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

Proof. The essential idea is to show that if a node P_i is assigned a task with a lower communication-to-computation ratio 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 throughputs. Thus, by a sequence of such swaps, any schedule can be transformed to one of the desired structure, without changing the fair throughput. See the Appendix A1 for a detailed proof. □

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-compute-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 of Section 4.5.

4. Demand-driven and Decentralized Heuristics

As shown in Section 3.1, given a tree-shaped platform and the set of all application parameters, we are able to compute an optimal periodic schedule. This approach suffers from several serious drawbacks. The first is that the period of the schedule is the least common multiple of the denominators of the solution of linear program (3). 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 [11, 6], 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 [6].

Since unlimited buffer space is unrealistic, we will only consider *demand-driven* algorithms. Each node has a 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. Figure 1 shows the pseudo-code for the scheduler thread. Note that line 2 says when there's room, the scheduler requests work from its parent. Because a request for work doesn't specify the type of the application, there must be enough room for any type task even after all previous outstanding requests are satisfied with tasks of the largest type. The "select" choices in line 5 depend on the particular heuristic used, and can be based on, for instance, the history of requests and task types it has received and the communication times it has observed for its children.

```
1: Loop
2:   If there will be room in your buffer, request work
     from 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 worker or a child's sched-
     uler) to assign work to and the type of application
     that will be assigned.
6:   If you have a task and a request 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
```

Figure 1. Demand-driven scheduler thread, run in each node

A second problem that some schedulers (including the one of Section 3.2) encounter is that centralized coordination may become an issue when the size of the platform grows beyond a certain point. It may be hard 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 (the solution to the linear programming problem), and four that are decentralized.

4.1. Centralized LP-based (*LP*)

If we know the computation power and communication speeds of all nodes in the distributed system, we can solve the linear programming problem (3.1) and tell each node the number of tasks of each type it should assign to each of its children each time period. Thereafter, no further global communication is required. Each scheduler thread uses a 1D load-balancing mechanism [4] to select a requesting thread and an application type.

The 1D load-balancing mechanism works as follows: if choice i should be made with frequency $f(i)$, and has already been made $g(i)$ 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 the *LP* heuristic would always converge to the optimal throughput, but we will see in Section 5.2.1 that it may not, primarily because of insufficient buffer space.

4.2. First Come First Served (*FCFS*)

The *FCFS* heuristic is a very simple and common decentralized heuristic. Each scheduler thread simply fulfills its requests on a *First Come First Served* basis, using the tasks it receives in order from its parent. The root ensures fairness by selecting task types using the 1D load-balancing mechanism with frequencies given by the applications' priority weights $w^{(k)}$. 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.

4.3. Coarse-Grain Bandwidth-Centric (*CGBC*)

This heuristic (*CGBC*) builds upon our previous work for scheduling a single application on a tree shaped platform [5, 3]. 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 highest priority. The scheduler always assigns tasks to the the highest-priority requester. Bandwidth-centric scheduling has been shown to have optimal steady-state throughput, both theoretically and, when the buffers are sufficiently large, in extensive simulations.

The idea of the *coarse-grain* heuristic is to assemble several tasks into a large one. More precisely, we build a macro-task out of $w^{(k)}$ tasks of type k, for each k. These are the units that are scheduled using the bandwidth-centric method. Thus, fairness is assured.

Unfortunately, even though bandwidth-centric scheduling can give optimal throughput of macro-tasks, the *CGBC* heuristic does not reach the optimal fair throughput. Indeed, Proposition 1 asserts that nodes with faster incoming links should process only tasks with larger communication-to-computation ratios. But since a macro-task includes tasks of all types, *CGBC* will send communication-intensive tasks to some low-bandwidth nodes.

4.4. Parallel Bandwidth-Centric (*PBC*)

The *parallel bandwidth-centric* heuristic (*PBC*) 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. Fairness is ensured by the root, which allocates tasks to its separate schedulers in the desired ratios.

In all our simulations, we enforce the one-port constraint for each scheduler thread. But for this *PBC* heuristic, we have not enforced the constraint globally across the schedulers. Thus, it is possible that a node will send as many as K tasks concurrently, one of each type. In this case, we do model the contention on the port, so the aggregate bandwidth doesn't exceed the port's limit. (Similarly, the node's processor can multitask between multiple tasks.) This gives the *PBC* strategy an unfair advantage over the other heuristics. In fact, it has been shown [11] that allowing *interruptible communication* (which is similar to concurrent communication) dramatically reduces the amount of buffer space needed to achieve optimal throughput.

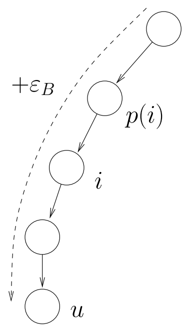
4.5. Data-Centric Scheduling (*DATA-CENTRIC*)

This heuristic is our best attempt to design a decentralized demand-driven algorithm that converges to the a solution of the linear program (3). The idea is to start from the bandwidth-centric solution for the most communication-intensive application and to progressively replace some of these tasks for more computation-intensive ones. Doing so, we come up with better values for the expected $\alpha_u^{(k)}$ and the expected $sent_{u\to v}^{(k)}$, which can in turn be used in the demand-driven algorithm of the Figure 1. These frequencies are continuously recomputed so as to cope with potential availability variations. The rest of this subsection is devoted to details of the *trading* operations.

We sort the task types by non-increasing communication-to-computation ratios. 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 applies the four operations described below, in the listed order of precedence. In the following, A (resp. B) denotes the application that currently has the highest (resp. lowest) weighted throughput relative to the target. Those operations attempt to increase the number of tasks of type B that are assigned, sometimes by reducing the number of A's.[2]

Communication Trading Suppose A has a higher communication-to-computation ratio than B, 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 can't keep up with the requests it receives from the grandchildren) then the parent can *substitute* some tasks of type A for type B (i.e. send some tasks of type B instead of tasks of type B to his under-utilized child). It should make the substitution in a way that keeps the communication time the same (i.e. trading them in the ratio of $b^{(B)}$'s A's for $b^{(A)}$ B's), and limited by the number that would make the weighted throughputs equal.

Gap filling Suppose that some bandwidth is not used and that a remote processor P_u could receive more tasks of an unfavored application.



Let ε_B denote the number of tasks of type B that this processor could handle. If we denote by CPU the CPU occupation of processor P_u, we have: $CPU = \sum_k \frac{\alpha_u^{(k)}.c^{(k)}}{c_u}$, and the following condition on ε_B has to hold true: $CPU + \varepsilon_B \frac{c^{(B)}}{c_u} \leqslant 1$. We also need to verify that there is enough free bandwidth along the path from the root node to P_u. Therefore for any node i along this path, we need the following condition on ε_B to hold true:

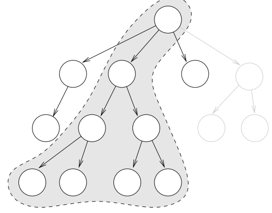
$$\underbrace{\sum_k \sum_j \frac{sent_{p(i)\to j}^{(k)}.b^{(k)}}{b_j}}_{bus\_occupation(p(i))} + \varepsilon_B \frac{b^{(B)}}{b_i} \leqslant 1$$

Lastly, to avoid over-reducing the imbalance between $\alpha^{(A)}$ and $\alpha^{(B)}$, we add the following constraint: $\alpha^{(A)} \geqslant \alpha^{(B)} - \varepsilon_B$. Therefore, we have:

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

2 In the following, we suppose without loss of generality that application characteristics have been scaled so that they all have the same priority weight.

Bus de-saturation The bus may have been saturated by tasks with a high communication-to-computation ratio. We may then still be using only workers with high communication capacity. In such a situation, the tree has to be *widened* (i.e. use additional subtrees) and the only way to do that is to reduce the amount of tasks of type A that are processed by the subtrees. The $\alpha_i^{(A)}$ and $sent_{i\to j}^{(A)}$ values of any node of the branch with the smallest bandwidth that process some tasks of type A are then scaled down by a factor of 0.9. This operation allows us to decrease the communication resource utilization and precedes *"Gap filling"* operations.



Task trading on the root At some point (when application A is processed only on the root node) we may have no choice but to trade ε_A tasks of type A for ε_B tasks of type B. Then we will have the following constraints: $\varepsilon_A \leqslant \alpha_0^{(A)}$, $\alpha^{(A)} - \varepsilon_A \geqslant \alpha^{(B)} + \varepsilon_B$ and $\varepsilon_B.\frac{c^{(B)}}{c_0} = \varepsilon_A.\frac{c^{(A)}}{c_0}$. Therefore, we have

$$\varepsilon_A = \min\left(\alpha_{root}^{(A)}, \frac{\alpha^{(A)} - \alpha^{(B)}}{1 + \frac{b^{(A)}}{b^{(B)}}}\right) \quad \text{and} \quad \varepsilon_B = \frac{c^{(A)}}{c^{(B)}}\varepsilon_A$$

The above operations are continuously performed (with the listed order of precedence) until we reach a satisfying balance, such as

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

The above operations may appear as needing a global knowledge. For example, it may seem at first sight that when performing a "*Gap filling*" operation, the master needs to know all informations on the the path connecting him to its remote descendant P_u. However, this operation in fact simply amount to compute a minimum along this path which can easily (and efficiently) be done by using a distributed propagation mechanism along this path, thus making the need of the master to know P_u irrelevant. The same kind of technique can be used for all other operations as they only imply descendants in a single subtree.

5. Simulation Results

5.1. Evaluation methodology

5.1.1. Throughput evaluation It is not at all obvious how to determine that a computation has entered steady-state behavior, 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 can then define the achieved throughput ρ_k for application k by:

$$\rho_k = \frac{N_k((1-\varepsilon)T) - N_k(\varepsilon T)}{(1-2\varepsilon)T} \quad \text{, where } 0 \le \varepsilon <, 0.5.$$

The ε is an artifact that lets us 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 solving linear program (3). Likewise, the minimum of the weighted experimental throughputs is called the *experimental fair throughput*.

5.1.2. 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 Appendix A2 for more details) in order to have wide trees rather than filiform (deep and narrow) ones. In our experiments, we generated trees of 5, 10, 20, 50 and 100 nodes. The maximum degree was 2, 5, or 15, and 10 trees of each configuration were generated. Thus, our test set comprised 150 trees in total.

Then we assign typical capacity, latency and CPU power 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 power ranged from 22.151 Mflops (an old Pentium Pro 200MHz) to 171.667 Mflops (an Athlon 1800). Bandwidth ranged from 110 kb/s to 7 Mb/s and latency from 6 ms to 10 s. Note that in the simulator that we are using (see Section 5.1.4), latency is a limiting factor as well as the link capacity for determining the effective bandwidth of a connection.

5.1.3. Application generation An application is mainly characterized by its *communication-to-computation ratio* (*CCR*). We decide that the smallest reasonable *CCR* was $CCR_{min} = 0.001$, which corresponds to the computation-intensive task of multiplying two 3500 × 3500 matrices. We also decided on an upper bound for *CCR* of 4.6, corresponding to the addition of two such matrices. In choosing application types, we chose 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 made all priority weights be 1.

5.1.4. Heuristic implementation The experiments were performed using the SimGrid simulator [16]. The simulator's performance is much more complex than the simplistic bandwidth and computational speed model used to design our heuristics. Therefore, the values of c_i and b_i values were measured from within the simulator and used to make the decisions in the algorithms of Section 4.

As explained in section 4.5, 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 using a larger number of tasks that this was always sufficient to reach the steady-state.

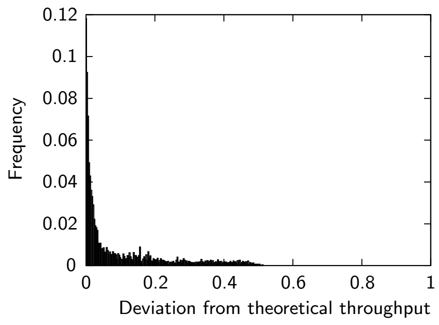


Figure 2. Deviation of experimental fair throughput from expected theoretical throughput

5.2. Case study

5.2.1. Theoretical versus observed throughput For the heuristics *LP*, *DATA-CENTRIC* and *CGBC*, we can easily compute an expected theoretical fair throughput. This allowed us to explore how implementation issues result in the experimental fair throughput differing from the expected theoretical fair throughput.

There are many reasons that the decentralized scheduling might have a smaller fair throughput than the corresponding theoretical one, such as the overhead of the request mechanism or a startup periods longer than the 10% we allowed for. It turned out that the major cause of inefficiency was the limit on the buffer size.

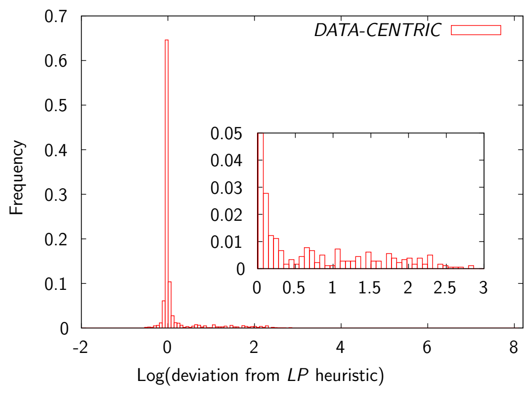
In general, our experiments assumed enough buffer space to hold 10 tasks of any type. In this case, Figure 2 depicts the experimental fair throughput deviation from the expected theoretical throughput for heuristics *CGBC*, *LP* and *DATA-CENTRIC*. All three heuristics exhibited a similar distribution, so they were combined in this figure. The average deviation is equal to 9.426%. However, when we increased the buffer size by a factor ten (and increased the number of tasks per application to 2000), the mean average deviation dropped to 0.334%.

Even though the larger buffer size led to much better throughput, we considered it unrealistic, and used size 10 in all other experiments.

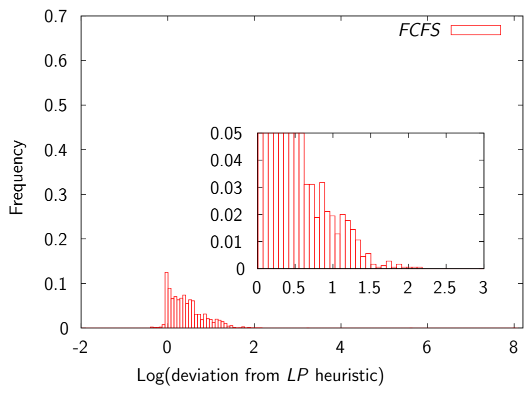
5.2.2. Performance of Heuristics Let us first compare the relative performances of our five heuristics (*FCFS*, *PBC*, *CGBC*, *LP* and *DATA-CENTRIC*). More precisely, for each experimental setting (i.e. a given platform and a given *CCR* interval), we compute the (neperian) logarithm of the ratio of 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). Therefore, a positive value means that *LP* performed better than the other heuristic. Figure 3 depicts the histogram plots of these values.

First of all, we can see that most values are positive, which illustrates the superiority of *LP*. Next, we can see on Figure 3(a) that *DATA-CENTRIC* is very close to *LP* most of the time, despite the distributed computation of the weights. However, the geometric average[3] 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 its performance was very bad (up to 16 times worse than *LP*). In contrast, *CGBC* (see Figure 3(d)) is much more stable since its worst performance is only two times worse than *LP*. Note that those failures happen on any type of tree (small or large, filiform or wide) and that the geometric average of these two heuristics are always very close to each other. We also have 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 (i.e. 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 low communication-to-computation ratio are mainly performed on the rightmost part of the tree while applica-

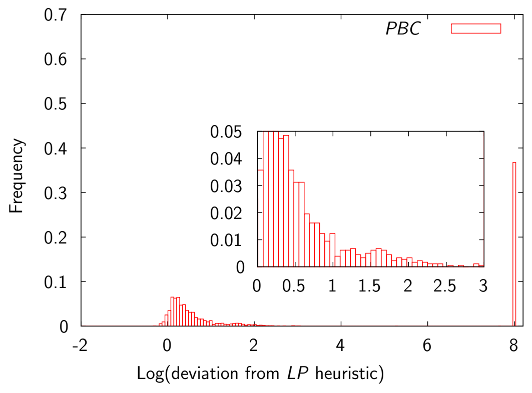
3 It is a well-known fact [15] that arithmetic average of ratios can lead to contradictory conclusions when changing the reference point. Therefore, we use a geometric average of ratios which is known to be closer to the general idea of *average ratio*.



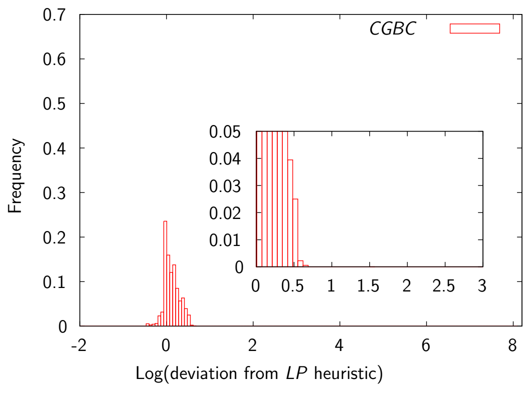
(a) Performances of *DATA-CENTRIC*



(b) Performances of *FCFS*



(c) Performances of *PBC*



(d) Performances of *CGBC*

Figure 3. Logarithm of the deviation from LP performances.

tions with a high communication-to-computation are mainly performed on the leftmost part, which is definitely not optimal on particular instances.

Unsurprisingly, *PBC* leads to very bad results. In many situations (more than 35%), 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.

Lastly, the geometrical average of *FCFS* is 1.564 and in the worst case, its performance is more than 8 times worse than *LP*. On the average, *FCFS* is therefore much worse than *LP*. On small platforms, the performances for *FCFS* and *CGBC* have the same order of magnitude. However, on larger ones (size 50 and 100), *CGBC* performs much better (geometrical average equal to 1.243) than *FCFS* (geometrical average equal to 2.0399).

6. Conclusion

In this paper, we present several heuristics for scheduling multiple applications on a tree-connected platform made of heterogeneous processing and communication resources.

Our contributions to this problem are the following:

- We first presented a centralized algorithm which, given the performances of all resources, computes an optimal schedule with respect to throughput maximization. We also have characterized a simple way of computing the optimal solution on single-level trees.
- However, on general platforms the centralized algorithm requires gathering information about the platform at a single location, which may be unrealistic for large-scale distributed systems, particularly when these parameters (bandwidths, processing power) may be constantly changing. Furthermore, the optimal schedule may require to have an arbitrary large number of buffers and may induce very large latencies. We have therefore concentrated on distributed algorithms and designed several decentralized heuristics using only a limited number of buffers.
- We have evaluated the efficacy of these heuristics using a wide range of realistic simulation scenarios. The results obtained by the most sophis-

ticated heuristics are quite reasonable compared to the optimal centralized algorithm.

Thus far, the best solutions rely on an explicit-rate calculation (using either a global centralized linear-based approach or a fully-distributed approach like in *DATA-CENTRIC*). It is a well-known fact in the network community [17] that max-min fairness is generally achieved by explicit-rate calculation (e.g. in ATM networks) and rather hard to achieve in a fully-decentralized fashion. Yet, fully distributed algorithms are known to realize other kind of fairness (e.g. proportional fairness for some variants of TCP). Adapting such algorithms to our framework is however really challenging as both communications and computations are involved. A promising approach would be to adapt the decentralized multi-commodity flow of Awerbuch and Leighton [1, 2] to our framework.

Last, as we have seen with the *PBC* heuristic, non-cooperative approaches where each application optimizes its own throughput lead to a particularly unfair Nash equilibrium [18, 12]. An other 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 [14, 13]. However in our situation, hierarchical resource sharing is rather hard to model, which renders such an approach quite challenging.

References

[1] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *FOCS '93: Proceedings of the 24th Conference on Foundations of Computer Science*, pages 459–468. IEEE Computer Society Press, 1993.

[2] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *STOC '94: Proceedings of the 26h ACM symposium on Theory of Computing*, pages 487–496. ACM Press, 1994.

[3] 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 Distributed Systems*, 15(4):319–330, 2004.

[4] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Trans. Computers*, 50(10):1052–1070, 2001.

[5] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS'2002)*. IEEE Computer Society Press, 2002.

[6] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Independent and divisible tasks scheduling on heterogeneous star-schaped platforms with limited memory. In *PDP'2005, 13th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 179–186. IEEE Computer Society Press, 2005.

[7] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[8] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[9] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[10] Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu`.

[11] L. Carter, H. Casanova, J. Ferrante, and B. Kreaseck. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2003*. IEEE Computer Society Press, 2003.

[12] F. Forgó, Jenö, and F. Szdarovsky. *Introduction to the Theory of Games: Concepts, Methods, Applications*. Kluwer Academic Publishers, 2 edition, 1999.

[13] D. Grosu and T. E. Carroll. A strategyproof mechanism for scheduling divisible loads in distributed systems. In I. C. S. Press, editor, *Proc. of the 4th International Symposium on Parallel and Distributed Computing (ISPDC 2005)*, 2005.

[14] D. Grosu, A. T. Chronopoulos, and M. Y. Leung. Load balancing in distributed systems: An approach using cooperative games. In I. C. S. Press, editor, *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 501–510, 2002.

[15] R. Jay. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., 1991.

[16] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[17] L. Massoulié and J. Roberts. Bandwidth sharing: Objectives and algorithms. *Transactions on Networking*, 10(3):320–328, june 2002.

[18] J. F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences USA*, 36:48–49, 1950.