HAL
open science

Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms

Pierre-Francois Dutot, Tchimou N'Takpé, Frédéric Suter, Henri Casanova

To cite this version:

Pierre-Francois Dutot, Tchimou N'Takpé, Frédéric Suter, Henri Casanova. Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms. IEEE Transactions on Parallel and Distributed Systems, 2008, 20 (7), pp.940-952. 10.1109/TPDS.2009.11 . inria-00347273

HAL Id: inria-00347273

https://inria.hal.science/inria-00347273v1

Submitted on 15 Dec 2008

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms

Pierre-François Dutot, Tchimou N'Takpé, Frédéric Suter and Henri Casanova

Abstract—Applications structured as parallel task graphs exhibit both data and task parallelism, and arise in many domains. Scheduling these applications efficiently on parallel platforms has been a long-standing challenge. In the case of a single homogeneous platform, such as a cluster, results have been obtained both in theory, i.e., guaranteed algorithms, and in practice, i.e., pragmatic heuristics. Due to task parallelism these applications are well suited for execution on distributed platforms that span multiple clusters possibly in multiple institutions. However, the only available results in this context are non-guaranteed heuristics. In this paper we develop a scheduling algorithm, MCGAS, which is applicable to multi-cluster platforms that are almost homogeneous. Such platforms are often found as large subsets of multi-cluster platforms. Our novel contribution is that MCGAS computes task allocations so that a (tunable) performance guarantee is provided. Since a performance guarantee does not necessarily imply good average performance in practice, we also compare MCGAS with a recently proposed non-guaranteed algorithm. Using simulation over a wide range of experimental scenarios, we find that MCGAS leads to better average application makespans than its competitor.

Index Terms—ixed parallelism, parallel task graph scheduling, performance guarantee, multi-cluster platform ixed parallelism, parallel task graph scheduling, performance guarantee, multi-cluster platform M

1 Introduction

Scientific simulations executed on parallel computing platforms can exploit two types of parallelism: *task parallelism* and *data parallelism*. A task-parallel application is partitioned into a set of tasks with possible precedence and communication constraints. A data-parallel application typically exhibits parallelism at the level of loops, i.e., iterations can be executed conceptually in a Single Instruction Multiple Data (SIMD) fashion. A way to expose increased parallelism, to, in turn, achieve higher scalability and performance, is to write parallel applications that use both types of parallelism, using what is often called *mixed parallelism*. With mixed parallelism applications are structured as *parallel task graphs* (PTGs), that is, task graphs of data-parallel tasks. PTGs arises naturally in many applications (see [1] for a discussion of the benefits of mixed parallelism and for application examples.) One well-known challenge for PTGs is *scheduling*, that is, making decisions for mapping computation and data transfers to platform components in a view to optimizing some performance metric. The vast majority of works that target the scheduling of PTGs use application execution time, or *makespan*, as the performance metric. Mixed parallelism adds another level of difficulty to the already challenging scheduling problem for task-parallel applications because data-parallel tasks are moldable, i.e., they can be executed on various numbers of processors, with more processors leading to faster task execution times. This raises the question of how many processors should be allocated to each data-parallel task. In other words, what is the best trade-off between running more concurrent data-parallel tasks with each fewer processors, or running fewer concurrent tasks each with more processors?

The most popular parallel computing platforms today are commodity clusters, which are therefore primary candidates for running PTGs. Most clusters consist of identical compute nodes (at least when they are initially put in production) and thus the question of scheduling PTGs on *homogeneous* platforms has been studied by many researchers. From a theoretical standpoint, although the scheduling problem is NP-complete, algorithms with performance guarantees, defined as the maximum ratio between the produced makespan and the optimal makespan, have been developed in [2], [3], [4], [5]. From a more applied standpoint, many non-guaranteed heuristics have been proposed and shown to lead to good average performance in practice [6], [7], [8], [9], [10], [11].

In spite of the abundance of deployed homogeneous clusters, heterogeneous platforms have received a lot of attention in the last decade. The primary motivation comes from improvements in network and middleware technology that have made it possible to aggregate several clusters over multiple institutions. These multi-cluster platforms, which are a form a *grid computing*, hold the promise of higher levels of scale and performance than possible with a single cluster. This is particularly true for task-parallel applications, which are less tightly

- *P.-F. Dutot is with Univ. Pierre Mendès-France, Grenoble 2 / LIG. UMR 5217 CNRS - INPG - INRIA - UJF, Grenoble 1 - UPMF, Grenoble 2.*
- *T. N'Takpé and F. Suter are with Nancy University / LORIA. UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1.*
- *H. Casanova is with the Information and Computer Sciences Department at the University of Hawai'i at Manoa and is supported in part by the National Science Foundation under award number 0546688.*

Manuscript received January 29, 2008; revised September 4, 2008; accepted December 8 2008.

coupled than purely data-parallel applications and can thus accommodate large inter-cluster network latencies (e.g., on wide-area networks). Consequently, many PTGs are well-suited to execution on multi-cluster platforms.

Multi-cluster platforms raise two challenges for the scheduling of parallel applications, and PTGs are no exception. First, these platforms are *heterogeneous* because they consist of clusters in different institutions. Second, they are *composite* and thus it is inadvisable to run data-parallel tasks across clusters, which adds an additional constraint when compared to the PTG scheduling problem for non-composite platforms. Developing PTG scheduling algorithms with performance guarantees on (multi-cluster) heterogeneous platforms is an open research question and previous work has instead focused on developing pragmatic heuristics [12].

In this paper we adapt theoretical results for PTG scheduling on homogeneous platforms to multi-cluster platforms. We address the two aforementioned challenges as follows. With respect to heterogeneity, we make the observation that there are deployed multi-cluster platforms (or significant subsets thereof) that exhibit low heterogeneity in terms of processor speed. Therefore, guaranteed scheduling algorithms developed for homogeneous platforms could lead to good results in our context. With respect to the composite nature of multi-cluster platforms we adapt recent theoretical results obtained for "hierarchical" clusters of Symmetric Multi Processors (SMPs) [13], noting that "collection of clusters" hierarchies are akin to "cluster of SMPs" hierarchies. More specifically, we make the following contributions:

- We develop the first practical implementation and experimental evaluation of a previously described task allocation algorithm that leads to a performance guarantee;
- We design a scheduling algorithm with a tunable performance guarantee for homogeneous multi-cluster platforms;
- We evaluate our scheduling algorithm in simulation to put its average performance in perspective with its performance guarantee; and
- We compare our scheduling algorithm with a recently published pragmatic heuristic for scheduling PTGs on multi-cluster platforms.

This paper is organized as follows. Section 2 details our platform and application models. Section 3 discusses related work. Section 4 presents our scheduling algorithm, which we evaluate in Section 5. Section 6 concludes the paper with a summary of our findings.

2 Platform and application models

2.1 Platform model

In this paper we base our platform model on a real-world multi-cluster platform, Grid'5000 [14], [15]. The goal of Grid'5000 is to build a highly reconfigurable, controllable and monitorable experimental platform to allow experimental parallel and distributed computing research. The platform consists of nine geographically distributed sites, aggregating a total of 5,000 CPUs, and is funded by the French ACI Grid incentive of the French Ministry of Research and Education. Each of the nine sites hosts at least one commodity cluster, and the number of processors per cluster ranges from around 100 to around 1,000. The architectures of these processors are AMD Opteron, Intel Xeon, Intel Itanium 2, or PowerPC.

Although the Grid'5000 platform is heterogeneous, it was established as a concerted effort with a goal of avoiding wide heterogeneity of processor performance. This may not be the case for other production grid platforms, in which it may not be possible to find a significant subset of the resources with low heterogeneity. By contrast, we can easily identify an "almost homogeneous" subset of Grid'5000. This subset comprises 545 processors distributed among six clusters. Table 1 summarizes the number of processors per cluster and the computing speed of the processors in each cluster, in GFlop/sec. These values were obtained with the High-Performance Linpack benchmark over the AMD Core Math Library (ACML) either with the original clock rates (on the Lyon, Nancy, Orsay, Rennes and Sophia sites) or by under-clocking the processors, which is done on the Lille site specifically to reduce the heterogeneity of Grid'5000. 545 processors, although amounting to only a little over 10 % of the overall platform, still represents a large homogeneous compute platform. There is therefore a strong motivation for attempting to use this almost homogeneous subset to the best of its potential for running PTGs, for instance by using sophisticated task allocation algorithms with performance guarantees.

Each cluster uses a Gigabit interconnect (GigaEthernet or Myrinet) internally, and all clusters are interconnected together by the wide-area RENATER Education and Research Network, a 10 Gigabit/sec network. Table 2 shows the inter-site latencies as measured on Grid'5000 in seconds. The simulations in this paper are based on the values given in Tables 1 and 2.

Table 1
An almost homogeneous subset of Grid'5000.

Site	Lille	Lyon	Nancy	Orsay	Rennes	Sophia
#proc.	53	56	47	216	99	74
Gflop/sec	3.336	3.254	3.379	3.388	3.364	3.258

Table 2
Inter-site network latencies (in msec.) on Grid'5000.

Site	Lille	Lyon	Nancy	Orsay	Rennes	Sophia
Lille	0.3	5.6	5.6	3.1	5.6	8.1
Lyon	5.6	0.3	5.6	3.1	5.6	3.1
Nancy	5.6	5.6	0.3	3.1	5.6	8.1
Orsay	3.1	3.1	3.1	0.3	3.1	5.6
Rennes	5.6	5.6	5.6	3.1	0.3	8.1
Sophia	8.1	3.1	8.1	5.6	8.1	0.3

2.2 Application model

A PTG application is modeled as a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \ldots, V\}$ is a set of vertices representing data-parallel tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges between vertices, representing communication between tasks. Each edge $e_{i,j}$ has a weight, which is the amount of data (in bytes) that task v_i must send to task v_j (we call v_j a *successor* of v_i and v_i a *predecessor* of v_j). Note that in addition to data communication itself, there may be an overhead for data redistribution, e.g., when task v_i is executed on a different number of processors than task v_j. Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task. Since data-parallel tasks can be executed on various numbers of processors, we denote by $T^k(v, p)$ the execution time of task v if it were to be executed on p processors of cluster C_k. In practice, $T^k(v, p)$ can be measured via benchmarking on each cluster for several values of p, or it can be calculated via a performance model. In this work we assume that $T^k(v, p)$ does not increases as p increases, i.e., using more processors for a task does not lengthen its execution time. The overall execution time of $\mathcal{G}$, or *makespan*, is defined as the time between the beginning of $\mathcal{G}$'s entry task and the completion of $\mathcal{G}$'s exit task.

We take a simple approach for modeling data-parallel tasks. We assume that a task operates on a dataset of d double precision elements (for instance a $\sqrt{d} \times \sqrt{d}$ square matrix). We arbitrarily assume that processors have at most 1 GByte of memory and thus $d \leq 121\,M$. We also assume that d is above $4\,M$ (if d is too small, the data-parallel task should most likely be fused with its predecessor or successor). The volume of data communicated between two tasks is equal to $8 \times d$ bytes. We model the computational complexity of a task, in number of operations, with one of the three following expressions, which are representative of common applications: $a \cdot d$ (e.g., a stencil computation on a $\sqrt{d} \times \sqrt{d}$ domain), $a \cdot d \log d$ (e.g., sorting an array of d elements), $d^{3/2}$ (e.g., multiplication of $\sqrt{d} \times \sqrt{d}$ matrices). For the first two types of complexity a is picked randomly between 2^6 and 2^9, to capture the fact that some of these tasks often perform multiple iterations. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three.

The above model leads to a range of communication-computation ratios that correspond to many typical computational tasks and real-world applications. More specifically, assume a 1Gbit/sec network (as the internal switches in our clusters) and 3.388 GFlop/sec processors (as the fastest processors in our platform). Our synthetic PTGs correspond to situations in which the total (sequential) time for performing all computations is between 1.1 and 42.8 times larger than the total (sequential) time for performing all data communications. Therefore, our experiments span the range from communication-intensive to computation-intensive applications.

While the above provides a model for sequential task execution we also need to account for parallel executions, i.e., for how task execution time varies with the number of processors. We use a simple model that is used extensively in the literature, thus allowing our results to be compared with previously published results consistently. This model is based on Amdahl's law [16] and specifies that a fraction γ of a task's sequential execution time is non-parallelizable. We pick random γ values uniformly between $0\,\%$ and $25\,\%$. With this "Amdahl model", an application task exhibits different execution times for different numbers of processors. We denote by ω_i the *work* of task v_i, i.e., the product of its execution time and of the number of processors allocated to it.

We consider applications that consist of 10, 20, or 30 data-parallel tasks. We use four popular parameters to define the shape of the DAG: width, regularity, density, and "jumps" The width determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2, 0.5, and 0.8 for width, and 0.2 and 0.8 for regularity and density. Finally, we add random "jumps edges" that go from level l to level $l + jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to "layered DAGs" [6]). We refer the reader to our DAG generation program and its documentation for more details [17]. Note that our DAG generation procedure is similar to ones used previously in the literature, for instance in [18]. It was also used to evaluate the HCPA scheduling heuristic [12], to which we compare the algorithm proposed in this paper.

Overall, we have $4^2 \times 3^3 = 432$ different DAG types. Since some DAG characteristics are random, for each DAG type we generate three sample DAGs, for a total of $1,296$ DAGs.

3 Related work

In this section we review related work, categorizing it with respect to the underlying platform model and referring both to theoretical results, such as guaranteed algorithms, and to pragmatic non-guaranteed heuristics, whenever applicable. The objective of all these algorithms is to minimize application makespan. This is the performance metric we use in this work as well.

3.1 Single homogeneous cluster

Early work in the area [19], [20] proposes an algorithm to compute optimal PTG schedules under strong assumptions, namely that all data-parallel tasks exhibit the same particular parallel performance behavior and that processor allocations can be fractional rather than integral. In the general case, a seminal result in the area of PTG scheduling from a theoretical standpoint is the guaranteed two-step algorithm proposed in [4]. In a first step the algorithm decides how many processors should be allocated to each task, which is done via a relaxed linear program minimization and which also results in fractional processor allocations. A rounding procedure is then used to obtain integral allocations [21]. In step two, the algorithm uses a simple list scheduling approach to map tasks to sets of processors. The guaranteed performance ratio is defined as the maximum ratio between the produced makespan and the optimal makespan. In [4] it is shown that the guaranteed performance ratio of this algorithm is $\sim$2.62 in the specific case of tree-shaped PTGs, and $\sim$5.24 in the general case. This result was improved in [5], leading to a $\sim$4.73 performance ratio in the general case. One of our contributions is that, to the best of our knowledge, we provide the first experimental evaluation of the approach in [4].

Several practical PTG scheduling algorithms based on heuristics have been proposed in the literature [6], [8], [9], [10], [11]. Like the guaranteed algorithms discussed earlier, the algorithms in [6], [8], [9], [10] proceed in two phases. A prominent algorithm is CPA (*Critical Path and Area-based scheduling*) [8], which aims at finding the best compromise between two quantities. The first quantity is the length of the *critical path*, i.e., the path in the PTG on which the sum of the edge and vertex weights is maximal. We denote the *length of the critical path* by C_{max}. The second quantity is the ratio of the *total work*, i.e., $W = \sum_{i=0}^{N} \omega_i$, and of the total number of processors, m. This ratio is then the *average work per processor*. The principle of the CPA algorithm is to start by allocating only one processor to each task. Therefore, initially C_{max} is larger than $\frac{W}{m}$. Then, at each iteration, CPA adds one more processor to the task belonging to the critical path that benefits the most from this 1-processor allocation increase. The allocation process stops when C_{max} becomes smaller than $\frac{W}{m}$. Indeed, the case $C_{max} = \frac{W}{m}$ corresponds to an optimal trade-off because both these quantities are lower bounds of the application makespan. Depending on application and platform characteristics, CPA may lead to excessively large allocations that can prevent the concurrent execution of independent tasks. Two algorithms address this limitation. MCPA [6], which is only applicable to layered PTGs, limits processor allocations to ensure that all the tasks in a level of the PTG can be executed concurrently. HCPA [12] employs a modified definition of the average work per processor to remove the bias induced by a large number of available processors and is applicable to any PTG. These last three algorithms all use a list-scheduling-based task mapping phase by which tasks are mapped to processors in order of decreasing "bottom level" (i.e., distance to the PTG's exit task), accounting for data communication and data redistribution costs. The iCASLB one-step algorithm in [11] was shown to lead to better performance than some two-step algorithms, including CPA, while maintaining reasonable complexity. This algorithm performs allocation and mapping simultaneously by iteratively increasing the allocations of tasks on the critical path, with a look-ahead mechanism to avoid being trapped in local minima, and a backfilling approach to improve the schedule.

3.2 Multiple homogeneous clusters

Scheduling algorithms with performance guarantees have been studied for a "hierarchy of homogeneous clusters", that is in fact a single cluster with identical nodes, where each node is a Symmetric Multi-Processor (SMP) and thus is a "cluster" of processors, where each cluster has the same number of processors. Using the work in [22] as a basis, in [13] Dutot has proposed extensions to the approach in [4] to accommodate multiple clusters. The key difficulty is that the execution time of a data-parallel task on a set of processors depends on the repartition of these processors among clusters. This difficulty is alleviated by enforcing a placement rule, which works as follows. Let p be the number of processors to be allocated to a data-parallel task, let s be the number of processors per cluster, and let q and r be the quotient and the remainder of the integer division of p by s ($p = q \times s + r$). An allowable placement must use q full clusters and r processors in a single cluster. This rule simply minimizes the number of different clusters used to run a data-parallel task.

The algorithm in [13] uses an allocation step similar to that in [4] for a single cluster and a specialized list-scheduling second step. However, the transition between the two steps is based on a differentiation between "small" and "large" tasks (depending on the number of allocated processors), and on constraints on the number of clusters that can run small tasks simultaneously. With the optimal choice for this differentiation and this constraint, the algorithm has an overall guarantee no worse than $\sim$5.64. This result holds for PTGs structured as trees, and a guarantee twice as large can be easily obtained in the general case. Note, however, that a lower guarantee in the general case could be achieved by leveraging the techniques proposed in [5].

To the best of our knowledge, no (non-guaranteed) scheduling heuristics were developed specifically for the case of multiple homogeneous clusters. However, the heuristics for multiple heterogeneous clusters reviewed in the next section are certainly applicable to multiple homogeneous clusters.

3.3 Multiple heterogeneous clusters

To the best of our knowledge no PTG scheduling algorithm with performance guarantees has been developed for heterogeneous (multi-cluster) platforms. Two heuristics have been recently proposed: HCPA (Heterogeneous CPA) [23] and M-HEFT [24]. HCPA extends the CPA algorithm [8] to heterogeneous platforms by using the concept of a reference cluster. Allocations on the reference cluster are translated into allocations on clusters containing processors with various speeds. M-HEFT extends the well-known HEFT algorithm for scheduling task-parallel DAGs [25]. M-HEFT performs list-scheduling by reasoning on average data-parallel task execution times for 1-processor allocations on all possible clusters. Weaknesses in both HCPA and M-HEFT were identified and remedied in [12]. In that paper, the authors perform a thorough comparison of both improved algorithms and find that although no algorithm is overwhelmingly better than the other, HCPA would most likely lead to schedules that would be preferred by the majority of users. HCPA was shown to achieve a good trade-off between application makespan and parallel efficiency (i.e., how well resources are utilized). In this work we compare our approach to this improved HCPA version.

4 A GUARANTEED ALGORITHM FOR HOMOGENEOUS MULTI-CLUSTER PLATFORMS

In Section 2.1 we noted that there are grid platforms, such as multi-cluster grids, in which significant subsets of the resources are almost homogeneous in term of compute speed. This provides the motivation for this paper, namely the investigation of a guaranteed algorithm for a homogeneous multi-cluster platform. Our work draws inspiration from the work in [13]. Recall that in that work the target platform is a homogeneous cluster of SMP nodes, while we consider a homogeneous collection of clusters. There are thus two key differences between the platform model in that paper and ours:

1) In [13] all nodes have the same number of processors (because they are homogeneous SMP nodes), but in this paper clusters can have different numbers of nodes (there are small clusters and large clusters).
2) In [13] data-parallel tasks are allowed to run over multiple nodes. By contrast, in this work we restrict a data-parallel task to run within a single cluster. Disallowing data-parallel tasks running over multiple clusters is sensible because of the cost of inter-cluster communication and this restriction is enforced in all previous work on the topic of PTG scheduling on multi-cluster platforms.

While the first difference above causes difficulties, the second one simplifies the scheduling problem. Indeed, with tasks running over a single cluster the placement rule defined in [13] and outlined in Section 3 is no longer necessary.

4.1 Fundamental Previous Results

Before presenting the details of our algorithm we recall two fundamental results that provide the basis of our approach. The first result, by Skutella [21], gives a linear program to find the task allocations that lead to the best possible trade-off between the length of the critical path and the average work per processor. The second result, by Lepère et al. [4], introduces the notion of bounding the number of processors allocated to each task to improve the performance ratio of the list scheduling.

4.1.1 The time-cost trade-off problem

The time-cost trade-off problem (see [26]) is very similar to the processor allocation problem we face when scheduling PTGs. As in the time-cost trade-off problem we have two lower bounds on the metric to be optimized: the length of the critical path and the average work per processor. Reducing the number of processors allocated to any task affects this trade-off in favor of a smaller average work, while it may increase the critical path. Conversely, increasing the number of processors used to compute a task is likely to shorten the critical path and to increase the average work per processor. The goal is to achieve the best trade-off between the two, i.e., minimizing their maximum. As a result, provided we are in a one-cluster scenario, we can reuse the approach in [21] for solving the time-cost trade-off problem directly for scheduling a PTG. We defined earlier $T^k(v,p)$ as the time it takes to complete task v when using p processors of cluster k. Since for now we consider only one cluster we shorten the notation to $T(v,p)$ in this section. The work used for task v on p processors is then $pT(v,p)$. There are m possible allocations for each task: execution time $T(v,p)$ and a work of $pT(v,p)$, for $p = 1,\ldots,m$. We thus have to solve a discrete optimization problem, i.e., finding the optimal trade-off by picking for each task a particular allocation among a finite set of possible allocations.

As with many problems, solving the discrete problem is strongly NP-hard, while solving a continuous version of problem is easy. The idea here is then to first solve a larger, but continuous problem, in which each task v is replaced by a set of $m-1$ "activities". Each activity has a continuous linear cost function defined based on execution time. To each activity v_i corresponds a variable $x_{v,i}$ verifying the following inequalities: $T(v,m) \leq x_{v,i} \leq T(v,i)$. The cost of activity v_i, denoted by $\omega_{v,i}$, is set to:

$$\omega_{v,i} = \frac{T(v,i) - x_{v,i}}{T(v,i) - T(v,m)}((i+1)T(v,i+1) - iT(v,i)).$$

Since $T(v,m)$ decreases as m increases, for all i smaller than m, $T(v,i)$ is no smaller than $T(v,m)$.

To keep track of the precedence constraints, we introduce variables s_v to ensure that no task starts before all its predecessors end. In short, for all $(u,v) \in \mathcal{E}$, $s_v \geq \max_i(s_u + x_{u,i})$.

To summarize, for any given critical path length $\bar{C}_{max}$, the minimal cost necessary to achieve it is found by solving the rational linear program defined by the following constraints:

$$\begin{aligned} \forall v, i, \quad & T(v,m) \leq x_{v,i} \\ \forall v, i, \quad & x_{v,i} \leq T(v,i) \\ \forall (u,v) \in \mathcal{E}, \quad & \max_i (s_u + x_{u,i}) \leq s_v \\ \forall v, i, \quad & s_v + x_{v,i} \leq \bar{C}_{max} \end{aligned}$$

And minimizing the cost function:

$$\sum_{v \in \mathcal{V}} \sum_{i=1}^{m-1} \omega_{v,i} \ .$$

Note that a fixed cost corresponding to the smallest possible total work, achieved when each task is allocated only one processor, has to be added to the above cost function in order to compute the true total work, which we denote by $\bar{W}$. We refer the reader to [21] for all details and justifications about the construction of the above linear program.

At this point, we have a way to pick any $\bar{C}_{max}$ value and compute the corresponding $\bar{W}$ value, with the goal of finding the optimal trade-off. The $\bar{C}_{max}$ and $\bar{W}$ values are continuous and not necessarily integers. Since $\frac{\bar{W}}{m}$ and $\bar{C}_{max}$ have opposite behavior, there are two possible scenarios. If one is always larger than the other, one can use straightforward extreme allocations (each task uses one processor if $\frac{\bar{W}}{m}$ is always larger, or all processors if $\bar{C}_{max}$ is always larger). Otherwise, an optimal trade-off can be approached by binary search. In the latter scenario, the values obtained are lower bounds of the optimal discrete trade-off, that is of the discrete values of the critical path length and of the total work so that the maximum of the critical path length and of the average work per processor is minimized. We denote these discrete values by C^*_{max} and W^*, respectively.

The work in [21] uses a rounding technique to turn the continuous solution $(\bar{C}_{max}, \frac{\bar{W}}{m})$ into a solution of the discrete problem with time and cost values $(C_{max}, \frac{W}{m})$, which are respectively lower than $\frac{1}{1-\mu} C^*_{max}$ and $\frac{1}{\mu} \frac{W^*}{m}$, where μ is a parameter that can be chosen arbitrarily between 0 and 1. Choosing an allocation with a processors for task v in the original problem amounts to setting all the $x_{v,i}$ to $T(v,m)$ with i lower than a and $x_{v,i}$ equals to $T(v,i)$ for i larger than or equal to a. The cost incurred in the linear program for task i with these $x_{v,i}$ values is then

$$\sum_{i<a} ((i+1)T(v,i+1) - iT(v,i)) = aT(v,a) - T(v,1) \ ,$$

which is the expected cost minus the aforementioned fixed cost corresponding to the smallest total work. When rounding the linear program's optimal solution, one must choose which $x_{v,i}$ will be set to $T(v,m)$ and which ones will be set to $T(v,i)$. This is done with respect to a threshold in the following way. If $\frac{T(v,i)-x_{v,i}}{T(v,i)-T(v,m)}$ is lower than or equal to μ, then $x_{v,i}$ is set to $T(v,i)$, reducing the work contribution $\omega_{v,i}$ of the corresponding activity to 0, while increasing the time dedicated to activity i:

$$T(v,i) - x_{v,i} \leq \mu(T(v,i) - T(v,m)) \leq \mu T(v,i) \ .$$

Therefore, $x_{v,i} = T(v,i) \leq \frac{1}{1-\mu} x_{v,i}$ and the time used for any activity is not increased by more than a factor $\frac{1}{1-\mu}$, hence $C_{max} \leq \frac{1}{1-\mu} C^*_{max}$.

Conversely, if $\frac{T(v,i)-x_{v,i}}{T(v,i)-T(v,m)}$ is greater than μ then $x_{v,i}$ is set to $T(v,m)$, reducing the time needed for the activity, while increasing $\omega_{v,i}$ to $(i+1)T(v,i+1) - iT(v,i)$. Since $T(v,i) - x_{v,i} > \mu(T(v,i) - T(v,m))$, straightforwardly $\omega_{v,i} > \mu\omega_{v,i}$, which means that the total work is not increased by a factor of more than $\frac{1}{\mu}$, hence $\frac{W}{m} \leq \frac{1}{\mu} \frac{W^*}{m}$.

We have thus computed discrete C_{max} and W values that are at most a factor $\frac{1}{1-\mu}$ and $\frac{1}{\mu}$ larger than the optimal discrete values, respectively.

4.1.2 Scheduling moldable tasks on a single cluster

Based on the linear programming approach in [21], the work in [4] focuses on how to schedule the tasks efficiently while preserving most of the allocations so that one can obtain a performance ratio derived from the lower bounds on the critical path and the average work per processor. The difficulty comes from the fact that the allocations are computed in a setting where an infinite number of processors can be used at the same time, since there are no constraints in the linear program on simultaneous execution of data-parallel tasks. With tasks with different execution times there is no simple geometrical transformation to transform a schedule for an unbounded number of processors into one for a fixed number of processors. The schedule has to be reconstructed from scratch, only keeping the allocation information.

The algorithm proposed in [4] is derived from the classical list scheduling algorithm. However, list scheduling cannot be used directly as it can be arbitrarily far from the optimal schedule. Consider for example an instance with m pairs of tasks where the first task has to be executed on all processors for a very short amount of time, while the second task has to be scheduled afterwards on a single processor for a long time. The worst case for list scheduling is to schedule all pairs one after the other, while the optimal is to schedule all the first tasks, and then all the second tasks in parallel resulting in a schedule without idle time.

To avoid the problem of having lots of ready tasks requiring too many processors, the solution proposed [4] is to enforce a maximum number of processors per task. In the original article this limit was noted $\mu(m)$, where m is the number of processors of the cluster, however to avoid confusions with the threshold of the linear program, this limit is noted b in the rest of this paper.

Simply put, the algorithm inserts a bounding step between allocation (derived from the time cost linear-program with parameter μ set to $\frac{1}{2}$) and placement

(according to a list scheduling algorithm). This bounding step ensures the desired performance ratio of $3+\sqrt{5}$. The analysis can be summarized as follows. There are three different kinds of time intervals in the output schedule:

- T_1: intervals where at most $b-1$ processors are used;
- T_2: intervals where at least b and at most $m-b$ processors are used; and
- T_3: intervals where at least $m-b+1$ processors are used.

As in Graham's classical proof for the $2-\frac{1}{m}$ performance ratio [27], a bounding of the critical path and of total work can be made with respect to the length of these three intervals, as:

- during the first kind of interval no tasks has seen its allocation reduced to exactly b processors (otherwise at least b processors would be used),
- during the first and second time interval no task is ready to be scheduled since there are at least b idle processors and the placement algorithm is a list scheduling algorithm.
- we can give for each kind of time interval a lower bound on the number of processors used, namely 1 for T_1, b for T_2 and $m-b+1$ for T_3.

A straightforward calculation yields the optimal b depending on the total number of processors, and the performance ratio. See [4] for all details.

4.2 The MCGAS algorithm

We call our new algorithm MCGAS (Multi-Cluster Guaranteed Allocation Scheduling). MCGAS, like the algorithm in [13] for scheduling PTGs on clusters of SMPs, relies heavily on the works in [21] and [4]. The first step of the algorithm is the allocation phase from [21], which rounds off the solution of a rational linear program corresponding to a time-cost trade-off problem, as explained in Section 4.1.1. Let us denote by C^*_{max} and W^* the values of C_{max} and W that correspond to the optimal trade-off. The allocations produced in this first phase ensure that C_{max} and W are at most $1/(1-\mu)$ and $1/\mu$ as large as C^*_{max} and W^*, respectively, where μ is a parameter between 0 and 1. Note that this approach has been repeatedly presented in the theoretical literature over the last decade. One of our contributions in this paper is that, to the best of our knowledge, we present the first practical implementation of the time-cost trade-off linear program. Therefore, for the first time, we are able to evaluate its efficacy for application scheduling in practice.

Once the initial processor allocation is determined, the schedule is produced via a modified list scheduling algorithm as in [4]. As explained in Section 4.1.2, we perform a bounding of task allocations so that these allocations are at most b, where the value of b is to be defined. A large value favors data parallelism, while a small value favors task parallelism. The goal for setting b to a value lower than, say, the number of processors of the largest cluster is to avoid ill-advised stalling of the critical path. Indeed, the allocation phase of MCGAS, albeit leading to a performance guarantee, does not attempt to balance data and task parallelism, and may thus lengthen the critical path in ways that could be avoided. Once all allocations have been bounded, tasks can then be mapped to processors using a list-scheduling algorithm. Figure 1 summarize the steps of the MCGAS algorithm.

Input: $G=(V,E)$, $T(v,n)$ for each $v \in V$ and n, μ, b.
Output: An allocation for each task and a schedule.
Steps:

1) Construct an instance of the continuous time-cost trade-off problem, based on G and $T()$.
2) Solve the continuous time-cost trade-off problem to obtain the optimal continuous trade-off.
3) Round off the solution of the continuous problem, so that C_{max} is a factor $1/(1-\mu)$ from optimal and W is a factor $1/\mu$ from optimal, while computing corresponding integer task allocations.
4) Bound all allocations to be at most b.
5) Use any list scheduling algorithm to schedule the tasks on the platform.

Figure 1. Main steps of the MCGAS algorithm

The efficacy of the scheduling algorithm and the performance guarantee are both contingent upon a good choice for the values of the μ and b parameters, as seen in the next section. It is important to note that the performance guarantee in MCGAS, which is detailed in the next section, does not account for communication between tasks. This is also the case for previously proposed guaranteed algorithms for homogeneous non-composite platforms [2], [3], [4], [5]. We leave the investigation of a guarantee that takes communication into account outside the scope of this paper.

4.3 Finding the best parameters for MCGAS

In this section we compute MCGAS's performance guarantee as a function of μ and b. We then show how to set the values of these two parameters so that the performance guarantee is as tight as possible.

We consider a multi-cluster homogeneous platform. Let n be the number of clusters in the platform, and p_i, $i=1,\ldots,n$, the numbers of processors in these clusters. We refer to p_i as the *size* of cluster i. Without loss of generality we assume that the clusters are sorted by non-increasing sizes ($p_1 \geq p_2 \geq \ldots \geq p_n$), so that p_1 is the size of the largest cluster.

Given the clusters sizes, we define

$$S = \sum_{i=1}^{n} \max\left(0, p_i - b + 1\right) ,$$

which is a quantity that we will use in what follows. Intuitively, S represents the minimum number of allocated processors so that no cluster has b idle processors.

For a given PTG we can categorize each time step in the resulting MCGAS schedule into three kinds of time intervals according to the following rules:

- T_1: intervals where at most $b-1$ processors are used;
- T_2: intervals where at least b and at most $S-1$ processors are used; and
- T_3: intervals where at least S processors are used.

The definitions of these intervals are adapted from those used in [4] (see Section 4.1.2), and use our newly defined constant, S. For the sake of simplicity we use t_i to denote the sum of the lengths of all intervals of type T_i.

The goal of this classification is to bound the contribution of each time step to C_{max} and to W. We know from [21] that after the rounding phase C_{max} is at most $1/(1-\mu)$ larger than C^*_{max}, and that W is at most $1/\mu$ larger than W^*. After the allocation bounding step, during which tasks that were allocated more than b processors are reduced to exactly b processors, W does not increase. Indeed, a smaller processor allocation for a task does not increase the task's work because we assume that tasks have parallel efficiencies lower than 1. For the same reason, the reduction to b processors causes C_{max} to increase by at most a factor p_1/b.

During intervals of type T_1, no task has seen its allocation reduced to exactly b processors (since fewer than b processors are used). Therefore for each interval T_1 there is a task that is on the critical path and whose allocation has not been reduced during the allocation bounding step. During intervals of type T_2, there is at least one cluster where b processors are idle, which means that no task is ready to be scheduled, which means again that there is a task in each of these intervals that belongs to the critical path. However, in this case the task may have seen its allocation reduced from p_1 processors to b processors. With this reduced allocation the task's contribution to C_{max} is at least b/p_1. For intervals of type T_3, there is no cluster with at least b idle processors, which means that there might be an unscheduled ready task that is on the critical path.

Consequently, on the one hand C_{max} is not smaller than $t_1 + b/p_1 \times t_2$, and on the other hand W is larger than $t_1 + b \times t_2 + S \times t_3$. Since the schedule length, C_{max}, is the sum $t_1+t_2+t_3$, we can now write a complete set of inequalities leading to the performance guarantee for our algorithm, using C^*_{max} to denote the optimal schedule length:

$$\begin{aligned} C_{max} &= t_1 + t_2 + t_3 \,, \\ \frac{C^*_{max}}{1-\mu} &\geq C_{max} \geq t_1 + \frac{b}{p_1}\, t_2 \,, \\ \frac{C^*_{max}}{\mu} \sum_{i=1}^{n} p_i &\geq W \geq t_1 + b\, t_2 + S\, t_3 \,. \end{aligned}$$

Let us define $m = \sum_{i=1}^{n} p_i$, and introduce a new parameter $\alpha \in [0,1]$. This parameter does not have any concrete interpretation, but is used as an algebraic device to combine the two above inequalities. More specifically, multiplying the first inequality by α, the second by $1-\alpha$, and adding them together, we obtain

$$\begin{aligned} C^*_{max} \geq & \left(\alpha(1-\mu) + \frac{(1-\alpha)\mu}{m}\right) t_1 \\ & + b\left(\frac{\alpha(1-\mu)}{p_1} + \frac{(1-\alpha)\mu}{m}\right) t_2 + \frac{(1-\alpha)\mu S}{m} t_3 \,. \end{aligned}$$

Let us now define β as the minimum of the three following quantities:

$$\begin{aligned} \beta_1(\alpha,\mu,b) &= \alpha(1-\mu) + \frac{(1-\alpha)\mu}{m}, \\ \beta_2(\alpha,\mu,b) &= b\left(\frac{\alpha(1-\mu)}{p_1} + \frac{(1-\alpha)\mu}{m}\right), \\ \beta_3(\alpha,\mu,b) &= \frac{(1-\alpha)\mu S}{m}. \end{aligned}$$

Using the fact that $C_{max} = t_1 + t_2 + t_3$, we obtain

$$C^*_{max} \geq \beta C_{max} \,.$$

The guaranteed performance ratio is thus equal to $1/\beta$, which is minimized when β is maximized. Finding a closed form for the α, b, and μ values that maximize β given the $(p_1, \ldots, p_n)$ values seems very challenging. But it turns out that it is possible to determine a good approximation of the solution.

The three quantities β_1, β_2, and β_3 are of the form $AX + BY$ with A equal to $\alpha(1-\mu)$, B equal to $(1-\alpha)\mu$, and with both X and Y greater than or equal to zero.

Lemma 1: Given two numbers X and Y greater than or equal to zero, and two parameters α and μ with values in $[0;1]$, the function $f(\alpha,\mu) = \alpha(1-\mu)X + (1-\alpha)\mu Y$ reaches its maximum when $\alpha = 1-\mu$.

Proof: Let $t = \sqrt{\alpha(1-\mu)}$. We prove that $(1-t)^2$ is larger than or equal to $(1-\alpha)\mu$:

$$\begin{aligned} (1-t)^2 &= 1 + t^2 - 2t = 1 + \alpha(1-\mu) - 2\sqrt{\alpha(1-\mu)} \\ &= 1 + \alpha - \alpha\mu - 2\sqrt{\alpha(1-\mu)} \\ &= 1 - \mu + \alpha + (1-\alpha)\mu - 2\sqrt{\alpha(1-\mu)} \\ &= (1-\alpha)\mu + (\sqrt{\alpha} - \sqrt{1-\mu})^2 \geq (1-\alpha)\mu. \end{aligned}$$

Since $t^2 = \alpha(1-\mu)$ and X and Y are greater than or equal to zero, we have $f(t, 1-t) \geq f(\alpha,\mu)$. Therefore, for any couple (α,μ) we can find another couple (α',μ') that verifies $\alpha' = 1-\mu'$ and $f(\alpha',\mu') \geq f(\alpha,\mu)$, which completes the proof.

Using Lemma 1, we can remove the α parameter from the equations:

$$\begin{aligned} \beta &= \min(\beta_1(\mu,b), \beta_2(\mu,b), \beta_3(\mu,b)) \\ &= \min\left((1-\mu)^2 + \frac{\mu^2}{m}, b\left(\frac{(1-\mu)^2}{p_1} + \frac{\mu^2}{m}\right), \frac{\mu^2 S}{m}\right) \end{aligned}$$

Let us compute the values of b and μ that maximize this quantity (recall that m and p_1 are characteristics of the platform and are fixed, while S is a piecewise linear function of b). Note that $\beta_1(\mu,b)$ depends only on μ. Also, β_1 decreases from 1 to $1/m$ when μ increases from 0 to 1, and β_3 increases from 0 to S/m when μ increases from 0 to 1. Therefore, the largest minimum of β_1 and

β_3 is achieved when $\beta_1(\mu, b) = \beta_3(\mu, b) = \beta_{1,3}$, that is when $(1-\mu)^2 = \mu^2(S-1)/m$. β is then maximized when $\beta_2(\mu, b)$ is equal to $\beta_{1,3}$, that is when $b(S-1+p_1) = Sp_1$. We obtain the best value for b as an integer approximation of the non-integer solution of this simple equation, and can then compute the best value of μ. Recall that by "best values" we mean the values that lead to the tightest performance guarantee.

With our particular target platform, described in Section 2.1, the above computation yields that the best β is reached when $b = 87$. This value is the smallest integer such that $b(S+p_1-1) > Sp_1$. In this case the best value for μ is $\mu = 1/(1+\sqrt{\frac{S-1}{m}}) \simeq 0.662$. The value of β is then $S/(\sqrt{S-1}+\sqrt{m})^2 \simeq 0.115$, which corresponds to a performance ratio $1/\beta$ close to 8.695.

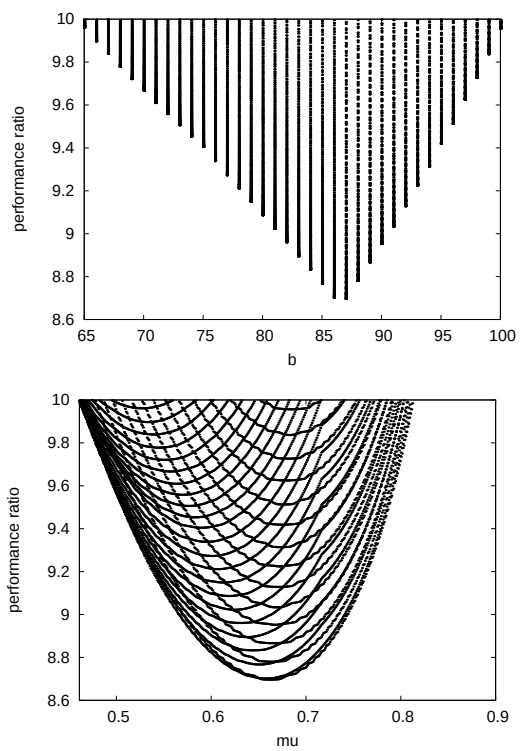


Figure 2. 2D projections of all $(\mu, b, 1/\beta)$ triplets with performance ratio $1/\beta$ lower than 10.

For given values of $(p_1, \ldots, p_n)$, in our case the values corresponding to the subset of the Grid'5000 platform described in Table 1, we can easily plot the different guaranteed performance ratios, $1/\beta$, each for given values of b and μ. For our platform configuration $1/\beta$ takes values in the interval $[8.696, +\infty[$ depending on b and μ. Figure 2 shows the two projections of all triplets $(b, \mu, 1/\beta)$ along the μ and the b axes, for performance ratios at most 10. The top graph shows the projection along the b axis. The bottom graph shows the projection along the μ axis. In both graphs we see that the performance ratio increases more sharply as b or μ become larger than their optimal values, and more moderately when they become smaller than their optimal values. Figure 3 shows the domain of the μ and b values in which one is guaranteed that the performance ratio is lower than 10.

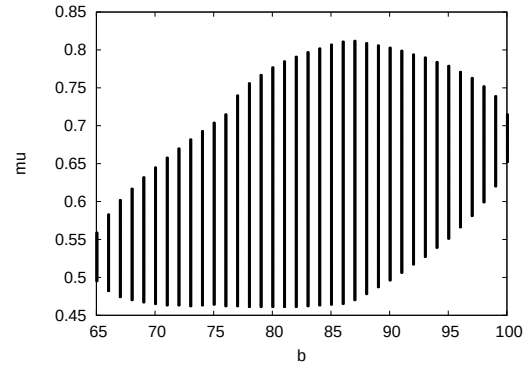


Figure 3. Domain of b and μ values for which MCGAS's performance ratio is lower than 10, for our particular platform confi guration.

These graphs, or similar graphs obtained for other platform configurations, provide good guidance for tuning the values of μ and b. Indeed, the values of μ and b that lead to the tightest performance guarantee may not lead to the best average application performance in practice. Therefore, one may wish to tune the μ and b values to ensure a reasonable performance guarantee while leading to good average observed performance over a range of relevant application configurations.

5 Simulation results

We use simulation for evaluating our proposed algorithm and for comparing it to previously proposed heuristics. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations (in a reasonable amount of time). We use the SimGrid toolkit [28], [29] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms. We use SimGrid v3.3-r5668. Our simulations are for the Grid'5000 platform as described in Section 2.1. They account for time taken by computation, data communication, and data redistribution operations (even though some of our algorithms may ignore data communication and data redistribution overheads when making scheduling decisions).

5.1 Processor allocation

The main strength of MCGAS, as discussed at length in Section 4, is that it computes a sound allocation of processors to data-parallel tasks, which is the basis for its performance guarantee. In this section we evaluate the quality of this allocation when compared to that of the allocation procedure used by the HCPA algorithm [12],

which improves upon that used by the seminal CPA algorithm [8].

We compare the quality of allocations as follows. For our 1,296 application configurations (see Section 2.2), we compute a processor allocation using MCGAS and HCPA. For each allocation we compute the length of its critical path (lower values mean better performance) and its total work (lower values mean lower resource consumption).

For each application configuration Figure 4 shows the length of the critical path achieved by the MCGAS allocation relative to that achieved by the HCPA allocation. We use the values $b = 87$ and $\mu = 0.66$ for MCGAS, which lead to the best performance guarantee. We show three curves, with a curve for each set of application configurations with a given number of tasks (10, 20, and 30). For each curve the data points are sorted by increasing value of the relative makespan. We see that across our application configurations the length of the critical path of the MCGAS allocation is at most $95.28\,\%$ of that achieved by HCPA. As the number of tasks increases, the relative critical path increases: MCGAS leads to critical paths $12\,\%$, $9\,\%$, and $8\,\%$ shorter than HCPA on average across application configurations with 10, 20, and 30 tasks, respectively.

Figure 5 is similar to Figure 4 but plots the total work of the allocations computed by MCGAS relative to that of allocations computed by HCPA. We see that overall MCGAS leads to allocations that consume more resources that HCPA. This is because HCPA was designed to limit resource consumption explicitly, as explained in [12]. Therefore, unlike MCGAS, HCPA tends to trade off shorter critical path length for lower resource consumption (which is the reason for the trends in Figure 4). However, we can see that MCGAS consumes fewer resources relatively to HCPA as the number of application tasks increases: MCGAS consumes $110\,\%$, $57\,\%$, and $35\,\%$ more resources than HCPA on average across application configurations with 10, 20, and 30 tasks, respectively.

Similar plots for application parameters other than the number of tasks (i.e., width, density, regularity, jumps, and complexity, which impacts the communication-computation ratio), not included here, show that the critical path length and total work of MCGAS, relative to that of HCPA, do not depend significantly on these parameters.

Ultimately, we wish to assess whether allocations produced by MCGAS are inherently better than those produced by HCPA in a view to minimizing application makespan. The length of the critical path and the total work divided by the number of processors are two lower bounds of application makespan. Therefore, for a given application configuration, we compute the maximum of these two lower bounds, which we term $\mathcal{M}$, as computed by MCGAS and by HCPA. A lower value of $\mathcal{M}$ indicates a better *opportunity* to achieve a lower makespan. We have seen that the makespan for MCGAS is lower than

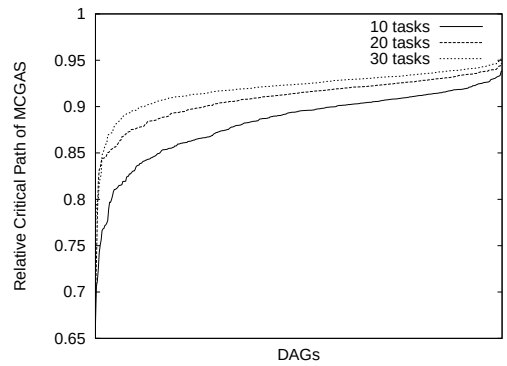


Figure 4. Relative critical path length of MCGAS ($\mu = 0.66$ and $b = 87$) compared to HCPA, for applications with 10, 20, and 30 tasks.

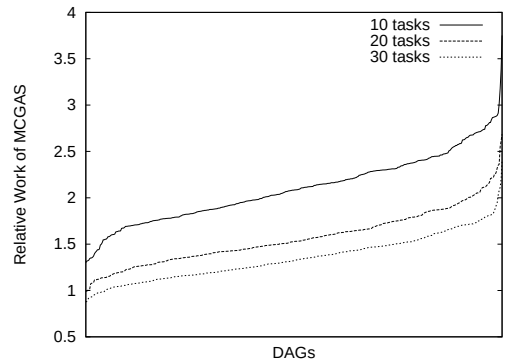


Figure 5. Relative total work of MCGAS ($\mu = 0.66$ and $b = 87$) compared to HCPA, for applications with 10, 20, and 30 tasks.

that of HCPA, and the total work for MCGAS is higher than that of HCPA. Therefore, it is not clear what the trend would be for the maximum of the two. It turns out that MCGAS achieves a lower $\mathcal{M}$ value in all cases across our experiments ($9.63\,\%$ lower on average, and at most $33.47\,\%$ lower). This means that when MCGAS exhibits higher total work than HCPA, then the makespan is larger than the average work per processor anyway, and thus determines the value of $\mathcal{M}$.

We also compared the allocations produced by MCGAS and HCPA for PTGs from real mixed-parallel applications. We used PTGs from the Strassen matrix multiplication and from the Fast Fourier Transform (FFT) application. Both are classical test cases for PTG scheduling algorithms [19], [30] and we refer the reader for instance to [31] for details on their PTGs. These PTGs are more regular than our synthetic PTGs, which are more representative of workflow applications with the composition of arbitrary operators in arbitrary ways. For each application we considered 10 different PTGs configurations. We found that on average MCGAS produces $\mathcal{M}$ values that are $39\,\%$ and $21\,\%$ shorter than HCPA for the Strassen matrix multiplication and the FFT application, respectively. The advantage of MCGAS is thus even larger than with our synthetic PTGs.

5.2 Task mapping

The task allocation computed by MCGAS provides a performance guarantee as long as mapping tasks to processors is done using list-scheduling. In the theoretical literature the default is to use a naïve First Fit (FF) strategy: for each ready task, simply map the task to one of the clusters that can start executing the task the earliest. In our case these candidate clusters are determined accounting for previous task mapping decisions but *ignoring* the (slight) heterogeneity of the platform and the cost of data communication/redistribution between tasks. In practice however, one is better advised to use a more sophisticated mapping strategy which, while not providing a tighter performance guarantee, should lead to better average performance. Comparing the average performance of FF and of this better strategy with a set of experiments is interesting. Let x denote the performance guarantee (i.e., the produced schedule is at most a factor x worse than optimal.) Then if one observes that the better strategy is on average a factor y better than the FF strategy, then it can be concluded that it is on average at most a factor x/y from optimal. The worst case performance is the same for both mapping strategies.

We experimented with a popular task mapping heuristic, Earliest Finish Time (EFT): for each ready task, map the task to one of the clusters that can complete the task the earliest. This computation accounts for previous task mapping decisions, for the (slight) heterogeneity of the platform, and for the cost of data communication and redistribution between tasks. We conducted experiments over all our application configurations for $b = 87$ and $\mu = 0.66$, values which lead to the best performance guarantee, i.e., 8.695.

As expected we find that the average performance of EFT is better than that of FF for all our experimental scenarios. The advantage of EFT increases for wider PTGs as the FF algorithm makes allocation decisions that end up hindering task parallelism, and we discuss average results based on PTG width. Specifically, we found that for PTGs with width 0.2, 0.5, and 0.8, EFT outperforms FF on average by $3.7\,\%$, $20.5\,\%$, and $34.2\,\%$, respectively. Therefore, we can conclude that *on average* EFT is at most a factor $8.695 \times .963 = 8.373$, $8.695 \times .795 = 6.913$, and $8.695 \times .658 = 5.721$ away from optimal for PTGs with width 0.2, 0.5, and 0.8, respectively. In the rest of the paper all results are presented using the EFT task mapping heuristic.

As in the previous section we evaluated MCGAS and HCPA on PTGs from real applications, namely the Strassen matrix multiplication and the FFT application. For the Strassen matrix multiplication we found that MCGAS produces makespans that are on average $22\,\%$ shorter than those produced by HCPA. But for the FFT application HCPA produces better results than MCGAS by $13\,\%$ on average. This is in spite of the MCGAS-produced allocation being inherently better than that produced by HCPA, as seen in the previous section. This phenomenon highlights a weakness of the EFT task mapping algorithm for the particular FFT PTGs. These PTGs are well-balanced in terms of communication and computation. An important characteristic is that some tasks produce large output data. However, EFT reasons only about task completion time, without "looking ahead" to see what amount of data will need to be communicated by a task after it has completed. Therefore, it can map tasks on different clusters, thereby forcing them to engage later in costly inter-cluster communications for their output data. When using the HCPA-computed allocations this weakness of EFT is not as noticeable because tasks are allocated fewer processors. Therefore, more tasks can fit on fewer clusters, thus saving communications. This observation is important because it motivates the development of more sophisticated task mapping heuristics to improve average application makespan, even when starting with good allocation decisions.

5.3 Trading off guarantee for performance

In this section we describe how the behavior of MCGAS can be tuned to trade off its performance guarantee (i.e., obtain a looser guarantee) for average performance (i.e., obtain a lower average makespan over our range of PTG configurations). Figure 6 plots the average makespan for different values of μ as b varies. A striking trend seen in

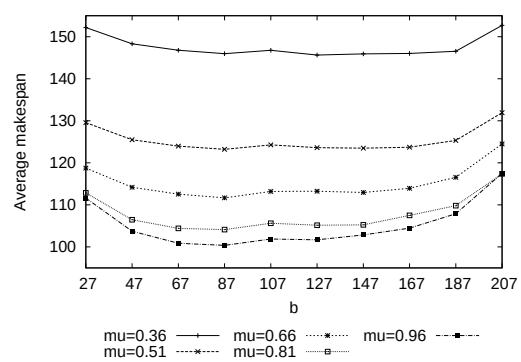


Figure 6. Evolution of the average makespan obtained with the MCGAS algorithm as b varies. Each curve is for a different μ value.

Figure 6 is the improvement in average makespan as μ increases beyond 0.66, value for which the performance guarantee of the allocation is minimal. The best value of μ leads to the tightest guarantee in terms of worst-case performance, while Figure 6 shows average performance over a particular population of PTG configurations. Nevertheless, one may have expected that values of μ that are far from its best value would produce allocations that are detrimental to the average makespan. One possible explanation is that our PTG configuration populations, although representative of real-world applications, is biased. In terms of DAG structure our PTGs do span

a wide range of characteristics. However, all our data-parallel tasks have reasonable parallel efficiencies ($\gamma < 0.25$), which are typical in real-world applications, but which could lead to the aforementioned bias. For this reason we also ran simulations for PTGs whose data-parallel tasks have poor parallel efficiencies ($\gamma > 0.5$), but we obtained similar results.

We conclude that indeed large μ values improve the average makespan, but one must be careful to consider the behavior of the average total work as well. We find that the average total work increases with μ. More specifically, in our experiments, we find that each time μ increases by .15, the average total work increases by $10\,\%$ to $20\,\%$. Therefore, while at first glance it seems that the best approach is to pick a μ value as large as possible, it leads to a less efficient use of the platform (implying higher cost in practice). Therefore, one should pick the largest μ value so that the performance guarantee is under some desirable value, thus attempting to strike a compromise between average makespan and efficiency while bounding the worst-case performance. In all that follows we pick the largest μ value so that the performance guarantee is lower than 10. Based on Figure 3, this value is $\mu = 0.81$.

Unlike for μ, picking values of b that are respectively larger or lower than the value that leads to the tightest performance guarantee is not desirable. Indeed, we see in Figure 6 that the best value of b ($b = 87$) leads to the best average makespan in practice. Using smaller values lengthens the critical path, while using larger values forces the allocation of most tasks to the largest cluster, preventing fruitful use of the other clusters.

5.4 Comparison of MCGAS and HCPA

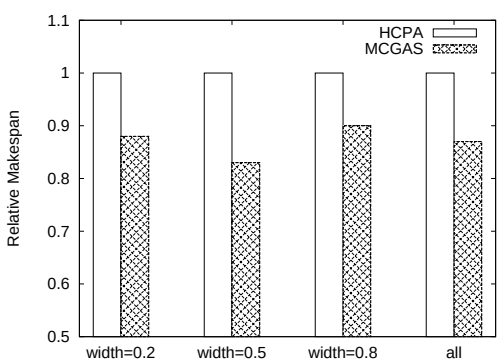


Figure 7. MCGAS vs. HCPA using $\mu = 0.81$, $b = 87$.

In this section we compare the average makespan achieved by MCGAS and HCPA. Figure 7 shows average makespans relative to the HCPA algorithm, over our range of application configurations. To the best of our knowledge, HCPA is the best previously published pragmatic algorithm for scheduling PTGs in multi-cluster platforms. The figure shows individual averages for PTGs with width 0.2, 0.5, and 0.8, and the overall average.

The main observation is that the MCGAS algorithm outperforms HCPA across the board. Grouping the PTGs by width, we see that this advantage is on average at most $17\,\%$, and $13\,\%$ when averaged over all PTGs.

We conclude that although a performance guarantee does not necessarily imply good average performance in practice, in this case configuring the algorithm with its best b value of 87 and with a high μ value so that the performance guarantee is under a particular bound (see previous section) leads to better average performance than that achieved by the best previously published non-guaranteed algorithm.

5.5 Comparison of scheduling times

While the results in the previous section demonstrate the superiority of MCGAS over HCPA, they are to be put in perspective with the time to compute the schedule. MCGAS involves solving a linear program, which can be time consuming. In this section we compare the execution time of MCGAS to that of HCPA. We ran both algorithms on an Intel Xeon 1.86GHz processor, using the GNU Linear Programming Kit (GLPK) library for solving the linear program. Table 3 shows the average execution times in seconds for different PTG sizes (i.e., numbers of tasks), each averaged over 10 runs for 10 different PTG configurations. The table also shows the average sequential application makespan and the average makespan produced by both algorithms.

Table 3
Average execution times and resulting application makespans of the MCGAS and the HCPA algorithms, in seconds, for different PTG sizes.

PTG size	avg. seq. makespan	MCGAS		HCPA	
		exec. time	makespan	exec. time	makespan
10	682.00	24.87	57.83	0.02	68.08
20	1,378.00	99.91	103.62	0.07	119.36
30	2,100.00	237.05	139.64	0.15	156.26

From the table we can see that, expectedly, the execution times of both algorithms increases with PTG size. More striking is the fact that MCGAS is much more expensive than HCPA. For instance, for PTGs with 30 tasks, MCGAS takes about 1,600 times longer to compute a schedule than HCPA. Most of the MCGAS execution time is due to solving a rational linear program The linear program. must be solved several times to implement the binary search described in Section 4.1.1. The number of steps of the binary search is constant for a given precision. Polynomial-time algorithms are known for solving rational linear programs (e.g., Karmarkar's algorithm [32]), and thus the complexity of MCGAS is also polynomial. Note that in this work we use the GLPK toolkit to solve linear programs. GLPK uses the simplex method. While this method is known to exhibit low polynomial-time complexity in practice in spite of a theoretical exponential-time complexity [33].

Table 4
Evaluation of MCGAS when limiting the number of possible allocations for each task.

PTG size	All allocations		20 allocations	
	makespan	exec. time	makespan	exec. time
50	198.80	**829.53**	199.56	**7.06**
100	272.88	**4758.68**	277.36	**90.64**

And indeed, in our case, at least up to 30 data-parallel tasks, the execution time appears roughly quadratic.

While the results in Table 3 seem to indicate that MCGAS may not be applicable to large PTGs, there are three simple reasons why its relatively high execution time can be reduced or tolerated. First, commercial solvers, such as CPLEX [34], are known to compute results in times up to several orders of magnitude shorter than GLPK, and their use would thus greatly reduce the MCGAS execution time. Second, the results in the previous section and those in the table show that running HCPA instead would lead to a performance loss $13\,\%$ on average, depending on the PTG's width. The scheduling time is thus to be put in perspective with the expected order of magnitude of the makespan. Importantly, the time to compute the schedule does not depend on the duration of the application tasks (only on the structure of the PTG and on the number of tasks). Therefore, the higher MCGAS execution time may be well worth the additional expense if the application makespan is large due to long tasks. In this case, the time to compute the schedule is negligible compared to the application makespan, and the makespan improvement when using MCGAS instead of HCPA can be significant. For instance, consider the last row of Table 3 and assume that the application tasks were 100 times more time consuming. The average application turn-around time, i.e., the sum of the time to compute the schedule and the application makespan, would be approximately $237.05 + 139.64 \times 100 \simeq 14,199$ for MCGAS and $0.15 + 156.26 \times 100 \simeq 15,626$ for HCPA. Third, it is important to note that in many production scenarios a (good) schedule is reused for many executions of an application, thereby amortizing the cost of computing the schedule in the first place.

A simple technique to reduce the execution time of MCGAS is to limit the number of possible task allocations that are considered in the time-cost trade-off problem. This technique reduces the flexibility of the allocation procedure and degrades the initial performance guarantee of MCGAS. As an example, Table 4 shows average makespans produced by and execution time of MCGAS for PTGs with 50 and 100 tasks, for the original algorithm and for a modified version of it that considers only 20 possible configurations for each task (corresponding to 1, 2, 3, 4, 5, 7, 9, 11, 14, 17, 23, 29, 38, 48, 62, 79, 102, 130, 168, or 215 processors). We see on the table that the modified algorithm achieves an average makespan that is less than 2% larger than that achieved by the original algorithm, while exhibiting an average execution time roughly 117 faster for 50-task PTGs, and 52 faster for 100-task PTGs. We conclude that this modification makes MCGAS usable in practice for large PTGs at the expense of the performance guarantee. In this article we have focused on the original MCGAS algorithm, which provides a better guarantee that the modified algorithms. Consequently, we have only presented results for PTGs with at most 30 tasks, for which MCGAS runs in under 4 minutes on an Intel Xen 1.86GHz processor.

6 Conclusion

Guaranteed algorithms for scheduling applications structured as parallel task graphs (PTGs) have been developed in the case of a single homogeneous parallel computing platform, such as a cluster [2], [3], [4], [5]. However, PTGs are particularly well suited to execution on multi-cluster platforms. In this context, to the best of our knowledge, the only previously proposed algorithm is the non-guaranteed, but pragmatic, HCPA algorithm [12]. In this paper we set out to develop MCGAS, a PTG scheduling algorithm that provides a performance guarantee for multi-cluster platforms. These platforms pose two challenges: they are heterogeneous and composite. Indeed, they consist of processors with different speeds located in clusters of difference sizes. We have for now side-stepped the first challenge by noting that there are real-world multi-clusters platforms that comprise homogeneous or approximately homogeneous subsets. Therefore, schedules computed assuming a homogeneous platform can be effective on such subsets. To address the second challenge, we have developed a guaranteed task allocation procedure that is applicable to a platform that consists of clusters with different numbers of processors. This guarantee is tunable via two parameters. We have determined the values of these parameters that lead to the tightest performance guarantee both analytically and in practice. While having a performance guarantee is always desirable, it does not mean that the algorithm leads to good average performance in practice. Our key finding, however, is that MCGAS outperforms the non-guaranteed HCPA algorithm on average over a large range of application configurations.

References

[1] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the Benefits of Mixed Data and Task Parallelism," in *Symposium on Parallel Algorithms and Architectures (SPAA'95)*, 1995, pp. 74–83.

[2] J. Turek, J. Wolf, and P. Yu, "Approximate Algorithms for Scheduling Parallelizable Tasks," in *Symposium on Parallel Algorithms and Architectures (SPAA'92)*, 1992, pp. 323–332.

[3] W. Ludwig and P. Tiwari, "Scheduling Malleable and Nonmalleable Tasks," in *Symp. on Discrete Algorithms*, 1994, pp. 167–176.

[4] R. Lepere, D. Trystram, and G. Woeginger, "Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints," in *9th Annual European Symposium on Algorithms - ESA 2001*, ser. LNCS, Springer-Verlag, Ed., no. 2161, 2001, pp. 146–157.

[5] K. Jansen and H. Zhang, "An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints," *ACM Transactions on Algorithms*, vol. 2, no. 3, pp. 416–434, 2006.

[6] S. Bansal, P. Kumar, and K. Singh, "An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines," *Parallel Computing*, vol. 32, no. 10, pp. 759–774, 2006.

[7] V. Boudet, F. Desprez, and F. Suter, "One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication," in *17th Int. Parallel and Distributed Processing Symp.*, 2003.

[8] A. Radulescu and A. van Gemund, "A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling," in *15th Int. Conference on Parallel Processing (ICPP)*, Valencia, Spain, Sept. 2001.

[9] S. Ramaswany, "Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications," Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, 1996.

[10] T. Rauber and G. Rünger, "Compiler Support for Task Scheduling in Hierarchical Execution Models," *Journal of Systems Architecture*, vol. 45, pp. 483–503, 1998.

[11] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications," in *35th Int. Conf. on Parallel Processing (ICPP'06)*, 2006.

[12] T. N'takpé, F. Suter, and H. Casanova, "A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms," in *6th International Symposium on Parallel and Distributed Computing*, Hagenberg, Austria, July 2007.

[13] P.-F. Dutot, "Hierarchical Scheduling for Moldable Tasks," in *11th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 3648. Springer-Verlag, 2005, pp. 302–311.

[14] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Irena, "Grid'5000: a Large Scale and Highly Reconfigurable Experimental Grid Testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, Nov. 2006.

[15] Grid5000, http://www.grid5000.org.

[16] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS 1967 Spring Joint Computer Conference*, vol. 30, Apr. 1967, pp. 483–485.

[17] F. Suter, "DAG Generation Program," http://www.loria.fr/~suter/dags.html.

[18] Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[19] G. N. S. Prasanna and B. R. Musicus, "The Optimal Control Approach to Generalized Multiprocessor Scheduling," *Algorithmica*, vol. 15, no. 1, pp. 17–49, 1996.

[20] ——, "Generalized Multiprocessor Scheduling and Applications to Matrix Computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 6, pp. 650–664, 1996.

[21] M. Skutella, "Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem," *Mathematics of Operations Research*, vol. 23, no. 4, pp. 909–929, 1998.

[22] P.-F. Dutot and D. Trystram, "Scheduling on Hierarchical Clusters using Malleable Tasks," in *Symposium on Parallel Algorithms and Architectures (SPAA'01)*. ACM Press, 2001, pp. 199–208.

[23] T. N'Takpé and F. Suter, "Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms," in *12th International Conference on Parallel and Distributed Systems (ICPADS'06)*, Minneapolis, MN, July 2006, pp. 3–10.

[24] H. Casanova, F. Desprez, and F. Suter, "From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling," in *10th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 3149. Springer-Verlag, Aug. 2004, pp. 230–237.

[25] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[26] M. Vanhoucke and D. Debels, "The Discrete Time/Cost Tradeoff Problem: Extensions and Heuristic Procedures," *Journal of Scheduling*, vol. 10, no. 4-5, 2007.

[27] L. R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal of Applied Mathematics*, vol. 2, pp. 416–429, 1969.

[28] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *10th Int. Conference on Computer Modeling and Simulation*, Mar. 2008.

[29] SimGrid, http://simgrid.gforge.inria.fr.

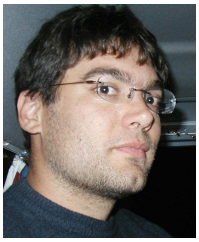
[30] H. Zhao and R. Sakellariou, "Scheduling Multiple DAGs onto Heterogeneous Systems," in *15th Heterogeneous Computing Workshop (HCW'06)*, Island of Rhodes, Greece, Apr. 2006.

[31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[32] N. Karmarkar, "A New Polynomial Time Algorithm for Linear Programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[33] D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, 3rd ed. Springer, 2008.

[34] CPLEX, http://www.ilog.com/products/cplex/.



Pierre-François Dutot Dr. Pierre-François Dutot is an Assistant Professor at the University Pierre Mendès France, Grenoble, France. His research interests include theoretical aspects of scheduling. He obtained his M.S. from the Ecole Normale Supérieure de Lyon, France in 2000 and his Ph.D. from the Institut National Polytechnique of Grenoble in 2004.

Tchimou N'Takpé Tchimou N'takpé is a Ph.D. student at Nancy University, France within the AlGorille INRIA project team. He received his M.S. in Engineering from Ecole Nationale d'Electricité et de Mécanique, Nancy, France, in 2005. His current research interest is scheduling mixed parallel applications.

Frédéric Suter Dr. Frédéric Suter is a CNRS junior researcher at the IN2P3 Computer Center in Lyon, France since October 2008. His research interests include scheduling, grid computing and platform and application simulation. He obtained his M.S. from the Université Jules Verne, Amiens, France in 1999 and his Ph.D from the Ecole Normale Supérieure de Lyon, France in 2002. Prior to joining the CNRS, he was an assistant professor at Nancy University and member of the AlGorille INRIA project team.

Henri Casanova Dr. Henri Casanova is an Associate Professor in the Information and Computer Science Dept. at the University of Hawaii at Manoa. His research interests are in the area of parallel and distributed computing. In particular, his research emphasizes the modeling and the simulation of platforms and applications, as well as both the theoretical and practical aspects of scheduling problems. He obtained his B.S. from the Ecole Nationale Supérieure d'Electronique, d'Electrotechnique, d'Informatique et d'Hydraulique de Toulouse, France in 1993, his M.S. from the Universite Paul Sabatier, Toulouse, France in 1994, and his Ph.D. from the University of Tennessee, Knoxville in 1998.