J. Parallel Distrib. Comput. 70 (2010) 1193–1203

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

On cluster resource allocation for multiple parallel task graphs

Henri Casanova[a], Frédéric Desprez[b], Frédéric Suter[c,*]

[a] Department of Information and Computer Sciences, University of Hawai'i at Manoa, USA
[b] LIP UMR 5668, ENS Lyon, INRIA, CNRS, UCBL, University of Lyon, France
[c] IN2P3 Computing Center, CNRS/IN2P3, Lyon-Villeurbanne, France

ARTICLE INFO

Article history:
Received 14 July 2009
Received in revised form
11 February 2010
Accepted 27 August 2010
Available online 26 September 2010

Keywords:
Multi-criteria scheduling
Resource allocation
Parallel task graphs
Cluster

ABSTRACT

Many scientific applications can be structured as parallel task graphs (PTGs), that is, graphs of data-parallel tasks. Adding data parallelism to a task-parallel application provides opportunities for higher performance and scalability, but poses additional scheduling challenges. In this paper, we study the off-line scheduling of multiple PTGs on a single, homogeneous cluster. The objective is to optimize performance without compromising fairness among the PTGs. We consider the range of previously proposed scheduling algorithms applicable to this problem, from both the applied and the theoretical literature, and we propose minor improvements when possible. Our main contribution is an extensive evaluation of these algorithms in simulation, using both synthetic and real-world application configurations, using two different metrics for performance and one metric for fairness. We identify a handful of algorithms that provide good trade-offs when considering all these metrics. The best algorithm overall is one that structures the schedule as a sequence of phases of increasing duration based on a makespan guarantee produced by an approximation algorithm.

1. Introduction

Scientific applications executing 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 constraints to form a *task graph*. A data-parallel application exhibits parallelism at the level of loops, so that iterations can be executed conceptually in a single instruction multiple data (SIMD) fashion. A way to expose increased parallelism, to achieve higher scalability and performance, is to design parallel applications that use both types of parallelism, using so-called *mixed parallelism*. With mixed parallelism, applications are structured as *parallel task graphs* (PTGs). A PTG is a directed acyclic graph (DAG) in which vertices represent tasks and edges represent precedence constraints and possible data dependencies between tasks. Each task is a data-parallel task. Each task is thus *moldable*, meaning that it can be executed using different numbers of processors. The number of processors allocated to a task is called the task's *allocation*. The task execution time is typically non-decreasing as the allocation increases. Fig. 1 shows two possible configurations of an example five-task PTG, each configuration corresponding to different allocations. PTGs arise naturally in many applications (see [8] for a discussion of the benefits of mixed parallelism and for application examples).

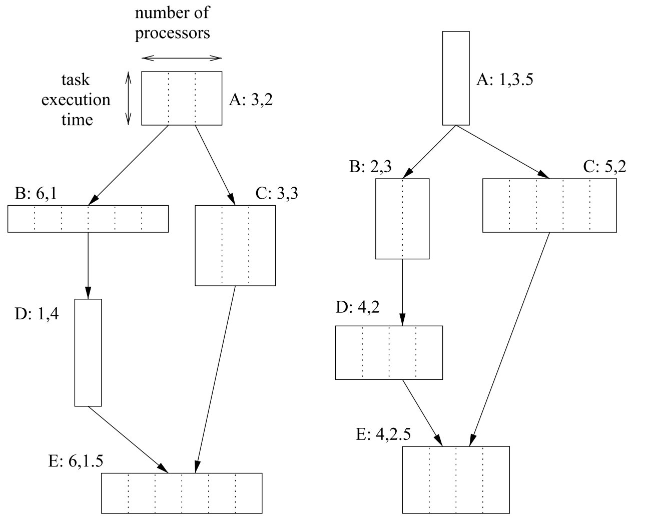
One well-known challenge for executing PTGs is *scheduling*, that is, making decisions for mapping computations to compute resources with a view to optimizing some performance metric. Mixed parallelism adds another level of difficulty to the already challenging scheduling problem for task-parallel applications because data-parallel tasks are moldable. 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, each with a smaller allocation, and running fewer concurrent tasks, each with a larger allocation?

Sharing of a commodity cluster for scientific application is traditionally done via batch schedulers [14]. Batch schedulers do not optimize user-perceived notions of performance and fairness [17,26]. Furthermore, most of them (the exception being OAR [4]) do not provide guidance for picking numbers of processors for tasks, which is a key question for PTG scheduling. For these reasons, we do not consider batch scheduling, but instead study the sharing of cluster resources among multiple PTGs in an off-line context, in which several PTGs have been submitted by different users and are ready to execute at the same time. The objectives are: (i) to minimize the overall makespan of the execution of all the PTGs; and (ii) to maximize the fairness among the PTG executions.

Three approaches have been proposed in the literature that relate to the above problem. In [31], Zhao et al. propose several

* Corresponding author.
E-mail address: Frederic.Suter@cc.in2p3.fr (F. Suter).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.08.017



(a) Example configuration.

(b) Example configuration.

Fig. 1. Example five-task PTG, with two possible configurations. Each task is labeled as $X : x, y$, where X is the task's name, x is the task's allocation, and y is the task's execution time.

techniques for scheduling multiple task graphs of sequential tasks. They either combine multiple task graphs into one and then use a standard task graph scheduling approach, or perform task-by-task scheduling over all tasks in all task graphs. In [20,21], N'takpé et al. propose a different approach, by which each PTG is simply given a subset of the processors and scheduled on this subset using a known PTG scheduling algorithm. The size of each subset is determined according to various criteria pertaining to the characteristics of the PTGs. Finally, in [11], Dutot et al. propose bi-criteria algorithms for scheduling independent moldable jobs, based on an approximation algorithm for optimizing the makespan [12]. Their work addresses a more general problem than the one addressed in this paper because PTGs are a special case of moldable jobs.

Our main contribution in this work is a comparison of all the above algorithms. This comparison is done in simulation, over a wide range of representative scenarios, using three metrics related to performance and/or fairness. Also, instead of using the algorithms exactly as they have been described in the literature, we have enhanced them with several common-sense improvements and, when required, modifications so that they are usable for our target problem.

This paper is organized as follows. Section 2 discusses related work. Section 3 details our platform and application models, and gives a precise problem statement. Section 4 describes all scheduling algorithms, which we evaluate experimentally in Section 5. Section 6 concludes the paper with a summary of our findings.

2. Related work

Several authors have studied the scheduling of multiple task graphs of sequential tasks onto heterogeneous platforms [31,9,16]. In [31], four algorithms are proposed that combine multiple task graphs into a single composite task graph, which is then scheduled using a standard task graph scheduling algorithm, and two algorithms that perform task-by-task scheduling over all tasks in all task graphs in a view to optimizing fairness. In [9], a two-level distributed scheduling algorithm for multiple task graphs is proposed. The first level is a WAN-wide distributed scheduler responsible for dispatching the different task graphs (viewed at this level as a single task) to several second-level schedulers that are LAN wide and centralized. The focus of that work is more on environment-related issues (such as machine failure rates and queue waiting times) than on scheduling concerns (such as promoting fairness among applications). In [16], the authors propose a hierarchical competitive scheduling heuristic for multiple tasks graphs. A restrictive assumption, which we do not make in this work, is that each application is responsible for its own scheduling and has no direct knowledge of the other applications. All the above focus on task graphs of sequential tasks and not on PTGs. Consequently, they do not address the difficult issue, which arises in PTGs, of how many processors should be given to each task. In this work we extend the algorithms in [31] to PTGs and include them in our evaluation.

Several algorithms have been recently proposed to schedule a single PTG. Most of these algorithms proceed in two phases. During a "resource allocation phase", they determine how many processors should be used for each task. Once all task allocations are determined, a "task mapping phase" is used to assign tasks to particular processors at particular times, ensuring that those processors are available to execute the task and that task precedence constraints are respected. The CPA (critical path and area) scheduling algorithm in [24] aims at finding the best compromise between the length of the *critical path* and the *average area* (which measures the sum of the processor × time area required by all tasks). One drawback of CPA is that, for some application and platform configurations, it produces allocations that are too large and reduce concurrency in a way that is detrimental to performance. The modified CPA (MCPA) algorithm in [2] addresses this drawback in the case of a PTG structured as a sequence of levels comprising independent tasks. Unlike CPA and MCPA, the iCASLB algorithm in [30] performs resource allocation and task 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 post-processing phase to improve the schedule. All three algorithms assume a homogeneous platform. The heterogeneous CPA (HCPA) algorithm in [19] extends CPA to heterogeneous platforms by using the concept of a homogeneous reference cluster and by translating allocations on that reference cluster into allocations on actual clusters comprising processors of various speeds. The M-HEFT algorithm in [6] extends the well-known HEFT algorithm for scheduling task graphs [29] to handle PTGs. Weaknesses in both HCPA (and thus in CPA) and M-HEFT were identified and remedied in [22], which performs a thorough comparison of both improved algorithms. In the case of a homogeneous multi-cluster platform, an algorithm to schedule a single PTG was recently proposed in [13]. This algorithm is based on a resource allocation algorithm that provides a performance guarantee, but has high computational complexity. Directly extending these heuristics and algorithms to schedule multiple PTGs simultaneously is an open question. In this work we use the improved HCPA algorithm in [22] for scheduling individual PTGs within specific subsets of the available processors.

To the best of our knowledge, the only previously published work that targets the scheduling of multiple PTGs is [21], which assumes a homogeneous, but multi-cluster, platform. The authors propose several algorithms that attempt to constrain the amount of resources that can be allocated to each PTG judiciously. We include these algorithms in our evaluation. In this work we restrain our study to the case of a single homogeneous cluster, which is the most commonly available type of parallel computing platform.

From the theoretical standpoint, the problem of scheduling multiple PTGs is a special case of the problem of scheduling independent moldable jobs, given that PTGs are inherently moldable. This problem has been studied for homogeneous clusters and a guaranteed algorithm for makespan optimization with a $3/2 + \epsilon$ approximation ratio is given in [12]. Particularly relevant for this work is the work in [11], which builds on the algorithm in [12] and proposes algorithms for solving a bi-criteria scheduling problem, with the two criteria being makespan and weighted average completion time. We build on the ideas in [11] to develop algorithms for scheduling PTGs rather than generic moldable jobs.

3. Problem statement

3.1. Platform and application models

A cluster consists of P *processors*. We use the term processor to refer to an individually schedulable compute resource. With this terminology, a "processor" may, in fact, be a physical compute node that is a multi-processor and/or multi-core computer. Processors are interconnected by a high-speed, low-latency network. A PTG is modeled as a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \mid i = 1, \ldots, V\}$ is a set of vertices representing data-parallel tasks, or *tasks* for short, and $\mathcal{E} = \{e_{i,j} \mid (i, j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges representing precedence constraints between tasks.

While we model task precedence constraints, we do not model any communication network or any data transfer between tasks, with the same rationale as in [1]. The algorithms we study generate schedules in which a task may complete well before the beginning of one or more of its successors. This affords much greater flexibility for fair sharing and efficient use of cluster resources for multiple simultaneous application executions, but precludes the use of synchronous network communication between tasks. Instead, the output data generated by a task must be stored temporarily until its recipient task begins execution. A simple way to store output data is to save them to files (e.g., via a parallel file system on a storage area network). We assume such an approach. The overhead of storing output data and of loading input data is comprised in each task's performance model as an overhead that depends on the input and output data size. Note that, in some schedules, a direct network communication between two tasks is possible when a successor task happens to start executing right after its predecessor completes. Nevertheless, we make the simplifying assumption that all data communication is handled via file I/O.

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 $T(v, p)$ the execution time of task v if it were to be executed on p processors. In practice, $T(v, p)$ can be measured via benchmarking for several values of p, or it can be calculated via a performance model. 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.

3.2. Metrics and problem statement

We consider the simultaneous execution of N PTGs on a cluster. The problem is to allocate resources to the tasks of these PTGs and to schedule them. We consider three metrics to quantify the quality of a schedule. For PTG $i = 1, \ldots, N$, we use $C^*_{\max_i}$ to denote the ideal execution time on the dedicated cluster, and $C_{\max_i}$ to denote the execution time in the presence of competition with the other PTGs.

We quantify the overall performance of the PTGs using the sum of completion times divided by the sum of ideal completion times, which we call the *scaled sum completion time*, and which is defined as

$$\sum_{i=1}^{N} C_{\max_i} \Big/ \sum_{i=1}^{N} C^*_{\max_i}.$$

While the sum completion time captures a notion of average performance, we also use a standard metric for the performance of the whole batch of PTGs, i.e., the *overall makespan*, defined as $\max_{i=1,\ldots,N} C_{\max_i}$.

One of our objectives is to evaluate the ability of scheduling algorithms to achieve fair schedules. A popular metric to evaluate the level of performance achieved by a job that competes with other jobs is the *stretch*, also called slowdown. In our case, jobs are PTGs, and the stretch of PTG i is defined as $C_{\max_i}/C^*_{\max_i}$. For instance, if a PTG could have run in 2 h using the entire cluster, but instead ran in 6 h due to competition with other PTGs, then its stretch is 3. This is the most widely accepted definition in the literature, with a lower value denoting better performance. Note, however, that in [31], whose algorithms we evaluate in this paper, Zhao et al. define the stretch as the inverse of the standard definition. In our description of the work in [31] we use the standard definition. It is understood that when we use terms like "increasing" or "smaller", in the original article Zhao et al. would have used "decreasing" or "larger".

A perfectly fair schedule can then be defined as a schedule in which all PTGs have the same stretch. One possibility is to define unfairness as the difference between the maximum stretch and the minimum stretch, or the average absolute value of the difference between the stretch of each PTG and the overall average stretch [31,21]. Another natural definition would be to define unfairness as the standard deviation or the coefficient of variance of the stretches. Yet another possibility, which we adopt in this work, is to quantify unfairness as the *maximum stretch*, defined as $\max_{i=1,\ldots,N} C^*_{\max_i}/C_{\max_i}$. If the maximum stretch is optimally minimized, then all PTGs have the same stretch, and fairness is optimal. Minimizing the maximum stretch has long been known to be a good approach to improve performance as well as fairness [3]. We select this metric because it is not completely agnostic to performance. Consider schedule A, in which all PTGs have a stretch of 1000, while a schedule B exists in which all PTGs can have stretches between 10 and 20. With the aforementioned fairness metrics, schedule A would be deemed preferable because the unfairness would be equal to 0. However, schedule B is preferable to all users, which is clearly indicated by the maximum stretch metric.

Note that the stretch can be used to define a performance metric as an alternative to our scaled sum completion time. For instance, performance can be quantified by the sum stretch or *average stretch* [3]. Nevertheless, we opt for scaled sum completion time, based on the following rationale. Consider a 100-processor cluster on which to schedule 101 independent, sequential jobs. 100 of these jobs run in 1 time unit, and one runs in ε time units. There are two reasonable types of schedule: either the task with execution time ε runs before a task with execution time 1 (schedule A), or it runs after such a task (schedule B). The standard average stretch definition gives $101 + \varepsilon$ for schedule A, and $101 + 1/\varepsilon$ for schedule B. As ε becomes small, in spite of the schedules becoming virtually identical, these stretches diverge and schedule B has an infinite average stretch. Using the scaled sum completion time definition, schedule A obtains $(100 + 2\varepsilon)/(100 + \varepsilon)$ and schedule B obtains $(101 + \varepsilon)/(100 + \varepsilon)$. As ε goes to 0, these two schedules have roughly equivalent stretches. We thus deem the scaled sum completion time more stable than the average stretch, which is why we use it in this work.

4. Resource allocation and scheduling algorithms

In this section, we describe the algorithms that we include in our study. Whenever applicable, we explain how we have improved algorithms proposed in the literature or how we have adapted them for our target scheduling problem.

4.1. Baseline algorithm

We consider a naïve algorithm that serves as a baseline comparator for all other algorithms. This algorithm, which we call SELFISH, operates as follows. For each PTG, SELFISH uses the HCPA algorithm in [22] to compute the allocation (i.e., the number of processors) for each task of the PTG. This is done assuming that the entire cluster is dedicated to the PTG's execution. The HCPA allocation computation proceeds as follows. Initially, each task is allocated only one processor. Task allocations are then increased iteratively. At each iteration, HCPA selects the task v, with current allocation p_v, on the current critical path that would benefit the most from one extra processor, i.e., the task v for which the ratio $T(v, p_v + 1)/(p_v + 1)$ is the smallest. The allocation of this task is then increased by 1 ($v_p \leftarrow v_p + 1$) and the process is repeated. HCPA uses a stopping criterion to decide when to stop increasing task allocations. The goal is to achieve an optimal trade-off between the length of the critical path and the average processor utilization. We refer the reader to [22] for the details of this stopping criterion.

Once an allocation has been computed for all tasks of all PTGs, these tasks are scheduled on the cluster using a simple list scheduling approach. Consider task $v_{i,j}$, i.e., task j in PTG i, and let $bl_{i,j}$ denote the task's bottom level. SELFISH sorts the tasks by decreasing $bl_{i,j}$ values, and then schedules each task, in this order, as early as possible. Once a full schedule has been produced there may be opportunities to reduce schedule fragmentation by moving task start times earlier. SELFISH takes advantage of such opportunities by using the same backfilling post-processing phase as that used in [30], inspired by the "conservative backfilling" technique used by batch schedulers [18]. Tasks are considered in the order in which they were scheduled, and each task is started as early as possible as long as no other task is delayed. We reuse this technique as a way to compact schedules at the last step of all the algorithms described hereafter, and simply call it "backfilling". We have found it to be beneficial for all our performance metrics for all algorithms.

One clear weakness of SELFISH is that it does not differentiate between "short" and "long" PTGs. Since it schedules all tasks of all PTGs together, by decreasing bottom level, the completion of a short PTG could be postponed, leading to a high stretch. Recall that for each PTG i we have applied HCPA assuming that the full cluster is dedicated to that PTG, leading to execution time $C^*_{\max_i}$. We propose two simple enhancements to SELFISH to ensure that tasks belonging to PTGs with short execution times are given higher priority. SELFISH_ORDER is similar to SELFISH, but instead of sorting tasks by $bl_{i,j}$ values it sorts them by increasing $C^*_{\max_i}$ values, and then by decreasing order of $bl_{i,j}$ values. This simply amounts to schedule short PTGs before long PTGs. SELFISH_WEIGHT instead sorts the tasks by decreasing $bl_{i,j}/(C^*_{\max_i})^2$. This heuristic attempts to weigh the bottom level of a task by the makespan of its PTG so as to give priority to tasks belonging to short PTGs. The use of the power 2 does not have a theoretical justification, but in our experiments led to much improved results. In all these algorithms all ties are broken randomly.

4.2. Algorithms based on Zhao et al.

In [31], Zhao et al. describe six algorithms for the scheduling of multiple task graphs of sequential tasks over heterogeneous platforms. These algorithms fall into two categories. The first category contains four algorithms, named C1, C2, C3, and C4, which combine the task graphs into a single composite graph. C1, C2, and C3 all combine the task graphs in a fork-join manner by making all their entry tasks successors of a new zero-weight entry task, and all their exit tasks predecessors of a new zero-weight exit task. C1 schedules the composite task graphs using a standard graph scheduling heuristic (the authors of [31] use either HEFT [29] or Hybrid.BMCT [25]). C2 and C3 partition the composite task graph into precedence levels, where all tasks within a level can be executed concurrently. Levels are then scheduled in sequence. C2 uses a standard independent task scheduling heuristic. C3 schedules tasks in a round-robin fashion in a view to increasing fairness. The C4 algorithm is markedly different in that it first determines the makespan of each task graph if it were to execute alone of the target platform using a standard task graph scheduling heuristic. Then shorter graphs are "grafted" onto larger graphs at judicious locations. This composite graph is then scheduled using a standard graph scheduling heuristic.

The two algorithms in the second category, F1 and F2, attempt to minimize unfairness at each scheduling decision. At each step, the algorithms estimate *current* stretches. These stretches are zero for task graphs with no scheduled tasks, which is the case of all graphs initially. The algorithms sort the task graphs first by descending stretches and then, for equal stretches, by decreasing maximum bottom level of tasks yet to be scheduled. A task from the first task graph in the list is then scheduled on a processor using a standard task graph scheduling algorithm. Therefore, at each step the algorithms attempt to schedule a task from the task graph with the worst current stretch. The F1 and F2 algorithms differ in the way they estimate the current stretch. After scheduling a task t of task graph $\mathcal{G}$, F1 computes $\mathcal{G}$'s new current stretch by dividing t's completion time by its completion time in the schedule computed assuming that $\mathcal{G}$ is alone on the platform. F2 instead updates the current stretch of all task graphs. Let ct be the completion time of task t. For each task graph, F2 finds the most recently scheduled task t' that executes at time ct, if it exists. F2 then determines what percentage of task t' is completed by time ct, and at what time the same percentage of task t' is completed in the schedule computed assuming that the task graph is alone on the platform. The current stretch for this task graph is then set to ct divided by this time.

The results in [31] show that F1 and F2 are superior to the other algorithms in terms of fairness, and essentially equivalent in terms of overall makespan. While C1 leads to the best makespans, it is the worst algorithm in terms of fairness. C1 is nevertheless interesting as it corresponds to a simple approach that is straightforward to implement. C4 is the best of the four algorithms that combine task graphs. Consequently, we include C1, C4, F1, and F2 in our study.

These algorithms are for scheduling task graphs of sequential tasks, while our goal is to schedule PTGs. Fortunately, it is possible to extend them to PTGs. In [31], each algorithm is based on a standard task graph scheduling algorithm to (i) compute the makespan of each task graph assuming it is alone on the platform; (ii) pick and schedule tasks; and (iii) estimate in-progress stretches for all task graphs. It turns out that we can use a standard PTG scheduling algorithm to the same end. We use the improved version of HCPA algorithm in [22]. Finally, we compact the produced schedules with the backfilling approach described in Section 4.1.

4.3. Algorithms by N'takpé et al.

In [21], N'takpé et al. propose to construct fair schedules for multiple PTGs by constraining the number of processors allocated to each PTG. More precisely, their resource allocation procedure ensures that no more than a fraction of the available processors in the platform is allocated to the tasks in a given precedence level of each PTG. The rationale is that the candidate ready tasks often belong to the same precedence level of a single PTG and thus can be executed concurrently. To prevent the postponing of tasks belonging to short PTGs (i.e., PTGs with short critical paths), a problem identified in [31], N'takpé et al. propose a mapping procedure that performs list scheduling only on ready tasks. In this way, even a short PTG can start at time 0. They name their family of algorithms constrained resource allocation (CRA).

N'takpé et al. consider several strategies to determine the resource constraint for each PTG. Three of these strategies lead to a good trade-off between fairness and overall makespan. The first, named CRA_NDAGS, simply gives an equal share of the resources to each PTG. The two other strategies account for specific characteristics of the PTGs. The CRA_WORK strategy accounts for the amount of computation for the PTGs by specifying the resource constraint of the ith PTG, P_i, as

$$P_i = \frac{1}{2N} + \frac{\omega_i}{2\sum_{j=1}^{N}\omega_j}, \quad (1)$$

where ω_i denotes the sequential work of the ith PTG (i.e., the sum of the sequential execution times of all its tasks).

The CRA_WIDTH strategy accounts for the maximum width of each PTG, i.e., the size of the precedence level that comprises the most tasks. A PTG with a large level should be able to exploit more task parallelism, and should therefore be given a larger allocation. Otherwise, this large level could become a performance bottleneck. The CRA_WIDTH strategy specifies the resource constraint of the ith PTG as

$$P_i = \frac{1}{2N} + \frac{width(i)}{2\sum_{j=1}^{N} width(j)}, \quad (2)$$

where $width(i)$ denotes the number of tasks in the largest level of the ith PTG. N'takpé et al. have shown that this last strategy is the fairest of the three.

For each of these algorithms, we also implement a WEIGHT variant that sorts the tasks in the same way as the SELFISH_WEIGHT variant of SELFISH, as described in Section 4.1. ORDER variants that sort tasks as SELFISH_ORDER are also possible, but our simulation results showed them to be outperformed by the WEIGHT variants. Therefore, given the sheer number of algorithms being evaluated in this work, we do not include them in our results.

We include these algorithms in our study but compact the schedules they produce using the backfilling approach described in Section 4.1.

4.4. Algorithms based on Dutot et al.

In [11], Dutot et al. propose algorithms for scheduling moldable independent jobs on a homogeneous cluster. All their algorithms rely on the $3/2 + \epsilon$ approximation algorithm in [12]. This algorithm computes an approximation of the optimal makespan, $C^*_{\max}$, computes an allocation for each moldable job (i.e., a number of processors), and produces a schedule. This schedule is structured as two phases, or "shelves", with jobs scheduled in either one of the two shelves (larger jobs in the first longer shelf, and smaller jobs in the second shorter shelf). The first algorithm in [11] simply uses the two-shelf schedule produced by the approximation algorithm. Two other algorithms are proposed that use only the allocations produced by the approximation algorithm and use two well-known list scheduling algorithms to schedule the jobs with these allocations. The first is LPTF (Longest processing time first), which gives priority to the job that has the longest processing time. The second is SAF (smallest area first), which gives priority to the job that has the smallest product of the number of processors allocated to it and its execution time.

The last algorithm proposed by Dutot et al. uses only the approximation of $C^*_{\max}$ computed by the approximation algorithm. It then partitions the time from 0 to $C^*_{\max}$ in K phases, or "shelves", where K depends on $C^*_{\max}$ and the smallest possible execution time over all jobs. By contrast with the schedule produced by the approximation algorithm, there can be more than two shelves, and the shelves increase in duration throughout the schedule. The goal is then to determine which jobs are scheduled within each shelf. This is done by solving a knapsack problem, via dynamic programming, for each shelf, from the smallest to the largest shelf. The goal is to maximize the "weights" of the jobs packed into each shelf, where the weights are those used for computing the weighted average completion time, which is one of the two criteria to optimize. The resulting schedule is then compacted via a number of optimizations (e.g., shuffle the order of shelves randomly, use a list scheduling heuristic to schedule jobs while respecting a given shelf order).

The work by Dutot et al. addresses a more general problem than ours, since PTGs are just one kind of moldable jobs. Indeed, since the tasks in a PTG are themselves moldable, a PTG can be executed on any number of processors p, with p varying from 1 to P. As a result, the algorithms in [11] produce coarse-grain schedules that cannot take advantage of the fine-grain structure of PTGs and lead to schedule fragmentation. We propose algorithms based on the ideas in Dutot et al., but that schedule PTGs and focus on optimizing makespan and fairness rather than makespan and weighted average completion time. We label this family of algorithms CAFM (coarse-grain allocation fine-grain mapping), since allocations are computed assuming generic moldable jobs, but task mapping is done with respect to individual PTG tasks.

The first step of all algorithms is to compute the execution time of each PTG assuming that p processors are available, with p varying from 1 to P. This is done using the improved HCPA algorithm in [22] to schedule the PTG on the given number of processors. In this way we obtain a specification of each PTG as a moldable job. We can then use the approximation algorithm in [12] to compute an approximation of $C^*_{\max}$, and an allocation for each job. Note that, unlike in [11], we do not attempt to reuse the two-shelf schedule produced by this algorithm for two reasons. First, the second shelf contains mostly smaller jobs, which is detrimental to fairness. Second, even if the two shelves were to be swapped, we propose below a K-shelf algorithm that subsumes the two-shelf approach.

Once allocations have been computed for each job, like in [11], we can use standard list scheduling techniques to schedule the jobs. Each job is then assigned a rectangular "box" in the schedule, which spans a number of processors and a number of time units. Within this box we can then schedule the individual tasks of the job, which is really a PTG. Recall that this is done using the improved HCPA algorithm in [22]. Once this is done for all PTGs, we obtain a schedule for all tasks of all PTGs. This schedule is likely very fragmented, and we compact it using backfilling, as explained in Section 4.1. For list scheduling we use the SAF and LPTF approaches as in [11]. However, we note that LPTF, unlike SAF, is likely detrimental to fairness since it gives priority to longer jobs and risks postponing the execution of shorter jobs. For this reason we also use a shortest processing time first (SPTF) approach. We obtain three algorithms: CAFM_LPTF, CAFM_SPTF, and CAFM_SAF.

The K-shelf idea in [11] is attractive from the perspective of fairness, as smaller jobs can be placed in smaller, earlier shelves. We reuse the algorithm from [11] to compute a K-shelf schedule for moldable jobs. One difference is that the packing of jobs into shelves solves a knapsack problem in which the weights of all the jobs are equal to 1, as opposed to an arbitrary weight. This is because our objective is to maximize an unweighted notion of fairness, rather than maximizing weighted average completion time. Note that, unlike in [11], we do not shuffle the shelf order as

Table 1
Cluster characteristics.

Cluster	grelon	grillon	chti	gdx
Number of processors	120	47	20	216
Gflop/s	3.185	3.379	4.311	3.388

shelves of non-decreasing durations promote fairness. Once a K-shelf schedule has been produced, as for the three CAFM algorithms described earlier, we schedule the tasks of each PTG within its box, and use backfilling to compact the schedule. We name this algorithm CAFM_K_SHELVES.

Finally, we propose an algorithm that combines the ideas from the CRA algorithms of N'takpé et al. (see Section 4.3) and those of Dutot et al. Recall that the algorithms of N'takpé et al. simply attempt to constrain the number of processors used for each PTG. While their algorithms use a number of heuristics to compute these constraints, it is possible to base them on the allocations computed by the approximation algorithm in [12]. These allocations are known to provide a strong guarantee on the overall makespan, and may therefore provide a good basis for producing a desirable overall schedule. Using the same procedure as for the other algorithms in this section, we execute the approximation algorithm and obtain an resource constraint for each PTG. Using this constraint, we can now compute an allocation for each task of each PTG using the improved HCPA algorithm in [22]. We can then schedule all tasks together, using the standard list scheduling approach of prioritizing tasks by decreasing bottom levels. Like for the other CAFM algorithms, we compact the schedule using backfilling. We name this algorithm CAFM_CRA.

Like for the algorithms described in the previous section, we also implement a CAFM_CRA_WEIGHT variant of CAFM_CRA that sorts the tasks in the same way as the SELFISH_WEIGHT variant of SELFISH, as described in Section 4.1. And here also the corresponding ORDER variant was found to be outperformed by the WEIGHT variant, and is not included in our results.

5. Experimental evaluation

5.1. Experimental methodology

We use simulation for evaluating the algorithms in the previous section. 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 v3.3.3 [7,27] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments, and it was specifically designed for the evaluation of scheduling algorithms. We explain hereafter how we instantiate the models described in Section 3.1 for the simulation experiments.

5.1.1. Platforms

We use configurations from four clusters in the Grid'5000 platform deployed in France [5,15]. Two of them, *grillon* and *grelon*, are located in Nancy, *chti* is located in Lille, and *gdx* is located in Orsay. Each cluster uses a Gigabit switched interconnect internally (100 μs latency and 1 Gb bandwidth). Table 1 summarizes the number of processors per cluster and the computation speed of the processors in each cluster, in GFlop/s. These values were obtained with the High-Performance Linpack benchmark over the AMD Core Math Library (ACML).

5.1.2. Applications

To instantiate the PTG model described in Section 3.1, we need to define specific models for execution times of data-parallel tasks and for the structure of the task graph.

We take a simple approach for modeling data-parallel task execution times. 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 GB of memory and thus $d \leq 121M$. We also assume that d is above 4M (if d is too small, the data-parallel task should most likely be fused with its predecessor or successor). 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., a 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. Beyond this model for sequential task execution, we also need to model parallel executions, i.e., how $T(v, p)$ varies with p. 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 and specifies that a fraction α of a task's sequential execution time is non-parallelizable. Recall that this non-parallelizable part comprises the overhead of the I/O necessary for data communication between tasks. We pick random α values uniformly between 0% and 25%. With this model, the task execution time strictly decreases as the number of processors increases.

We consider random PTGs that consist of 10, 20, or 30 data-parallel tasks. We use four commonly used parameters to define the shape of a PTG: width, regularity, density, and jumps. The width determines the maximum parallelism in the PTG, that is, the number of tasks per level. A small value leads to "chain" PTGs and a large value leads to "fork-join" PTGs. 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 PTG, 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. This leads to PTGs with mean maximum task parallelism of $V^{0.2}$, $V^{0.5}$, or $V^{0.8}$, where V is the total number of tasks, and coefficients of variance 20% or 80%. As a result, we generate PTGs that are close to chain graphs and PTGs that are close to fork-join graphs, with a spectrum of configurations in between. Furthermore we add random "jump edges" that go from level l to level $l + jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). We refer the reader to our graph generation program and its documentation for full details on the graph generation algorithm [28]. Generating three samples for each DAG configuration, and accounting to the four aforementioned computational complexity scenarios, we obtain a total of $3 \times 4 \times 3 \times 2 \times 2 \times 3 \times 3 = 1296$ different random DAGs.

While the above specifies a way to generate a population of synthetic PTGs, we also consider real PTGs from the Strassen matrix multiplication and from fast Fourier transform (FFT) applications. Both are classical test cases for PTG scheduling algorithms, and we refer the reader for instance to [10] for details on their structure. These PTGs are more regular than our synthetic PTGs, which are more representative of workflow applications that compose arbitrary operators in arbitrary ways. We consider FFT PTGs with 2, 4, and 8 levels (that is, 5, 15, and 39 tasks, and a maximum parallelism of 2, 4, and 8, respectively) All the Strassen PTGs have 25 tasks, and the maximum parallelism is 10. As

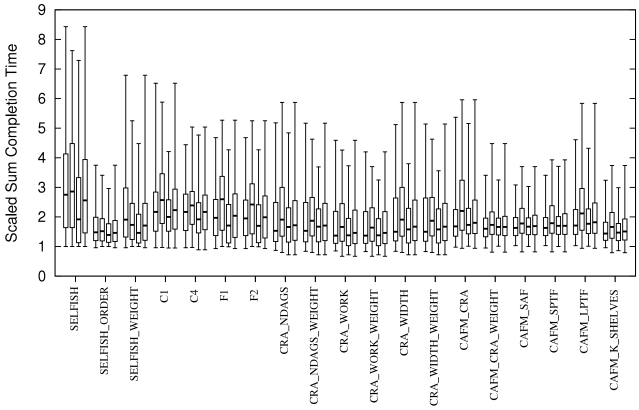


Fig. 2. Distribution of scaled sum completion time values for all algorithms in box-and-whiskers fashion. For each algorithm, the results are shown for random, Strassen, FFT, and all PTGs.

for the random DAGs, we consider four different computational complexity scenarios. This is to explore scenarios beyond those corresponding to the actual FFT and Strassen applications. For each FFT PTG configuration we generate 10 samples, for a total of $3 \times 4 \times 10 = 120$ PTGs. For each Strassen PTG configuration we generate 25 samples, for a total of $1 \times 4 \times 25 = 100$ PTGs.

Considering random, FFT, and Strassen PTGs, the number of tasks per PTG ranges between 5 and 39 tasks. This is representative of real-world workloads. For instance, the study in [23] reports on scientific workflow applications that are found in the workload of the Austrian Grid platform. Most of these workflows comprise around 30 tasks, 75% of them have fewer than 40 tasks, and only 5% of them have large numbers of tasks (higher than 200).

For each of our three classes of PTGs, we consider problem instances for $N = 2, 4, 6, 8$, and 10 concurrent PTGs. For each value of N, we generate 25 random sets of PTGs. Since we consider four different platforms, we have $25 \times 4 = 100$ instances of our scheduling problem for each class of PTGs, for a total of 300 instances. This amounts to a total of 1500 simulation scenarios, since we use five different values for N. Each algorithm is executed for each scenario.

5.2. Simulation results

In what follows, we first present results for all our algorithms for the scaled sum completion time metric (Section 5.2.1), the makespan metric (Section 5.2.2), and the maximum stretch metric (Section 5.2.3). We also present results that compare the algorithms for two metrics simultaneously (Section 5.2.4). We then study how our fairness metric, the maximum stretch, varies with the number of PTGs that are simultaneously scheduled (Section 5.2.5). We report one results regarding cluster utilization (Section 5.2.6). Finally, we discuss the execution times of the different algorithms (Section 5.2.7).

5.2.1. Scaled sum completion time distribution

Fig. 2 shows the distribution of scaled sum completion time values for all algorithms. The results are presented in box-and-whiskers fashion. The box represents the inter-quartile range (IQR), i.e., all the values comprised between the first (25%) and third (75%) quartiles, while the whiskers show the minimal and maximal values. The horizontal line within the box indicates the median, or second quartile, value. For each algorithm the figure displays four box-and-whiskers diagrams, from left to right: (i) results for random PTGs; (ii) results for Strassen PTGs; (iii) results for FFT PTGs; and (iv) results computed over all PTGs.

We first discuss the sensitivity of our results with respect to the three PTG populations. Focusing on median values, one observable trend is that, except for the algorithms in the SELFISH family, the scaled sum completion times are higher for the Strassen PTGs than for the random and FFT PTGs. The relative difference between the median value for two PTG populations is as high as 52.0% for F1 and 49.0% for SELFISH. For the other algorithm, this relative difference goes from 7.5% to 42.4%, and is 19.2% on average. In terms of absolute values, the difference between median values for different PTG populations for a single algorithm is on average 0.40. One important question is how the ranking of the algorithms varies according to different PTG populations. Still focusing on median values, we compute the rankings of the algorithms for each of the three populations and for all PTGs. The rank values range from 1 to 19 since there are 19 different algorithms. We find that the rank of an algorithm using the random, FFT, and Strassen PTG populations differs from the rank using all PTGs by, on average, 1.68, 1.57, and 2.10, respectively. The difference in ranking for an algorithm that is the ith best performing algorithm for a given PTG population and is the jth best performing algorithm considering all PTGs is computed as $|i - j|$. SELFISH_WEIGHT has the highest such differences in ranking (4, 5, and 6 for each of the PTG populations), but turns out to be consistently outperformed by SELFISH_ORDER and is not a contender for being the algorithm of choice. We conclude that, given the relatively small differences in ranking, using the results computed over all PTGs gives a reasonable view of the results. For the rest of this discussion, we consider only the fourth box-and-whisker diagram for each algorithm.

SELFISH leads to the worst results, with maximum scaled sum completion times up to 8.43. The SELFISH_ORDER variant leads to dramatically better results with a maximum of 3.75, a third quartile of 2.46, a second quartile of 1.71, a first quartile of 1.17, and a minimum of 0.96. The SELFISH_WEIGHT variant, although it improves on SELFISH, is significantly worse than SELFISH_ORDER, with, for instance, a maximum as high as 7.69 and a median at 1.71. We have mentioned that for algorithms in the CRA and the CAFM families the ORDER variants, not included here, were outperformed by the WEIGHT variants. This trend is reversed for algorithms in the SELFISH family, but our inspection of the schedules did not reveal any clear reason for this trend reversal. All algorithms based on the algorithms of Zhao et al. (see Section 4.2) are outperformed by SELFISH_ORDER. The best algorithm in the CRA family is CRA_WORK_WEIGHT, but its performance is only slightly better than that of CRA_WORK (the only differences are a third quartile lower by 0.05 and a maximum lower by 0.39). CRA_WORK_WEIGHT leads to results less consistent than those of SELFISH_WEIGHT: it has higher maximum and third quartile values, similar median value, and lower first quartile and minimum values. The best of the CAFM algorithms is CAFM_K_SHELVES, with CAFM_SAF and CAFM_SPTF close behind. Note that CARM_LPTF leads to much worse worst cases, with a maximum scaled sum completion time almost as high as 6. This confirms that LPTF schedules short PTGs late, thus increasing their completion times unnecessarily, as explained in Section 4.4. CAFM_CRA suffers from the same weakness. More generally, our results show that algorithms that explicitly schedule short PTGs first outperform the other algorithms in the same family that do not. The performance of CAFM_K_SHELVES is almost identical to that of SELFISH_ORDER for all the quartiles. We conclude that, for the scaled sum completion time metric, SELFISH_ORDER and CAFM_K_SHELVES are the best algorithms. We note, however, that several algorithms achieve reasonable performance with maximum values around 4 and median values under 2.

An interesting phenomenon, which occurs in 6% of our simulation scenarios, for algorithms in the CRA family, is that a PTG can experience a shorter makespan in a concurrent schedule

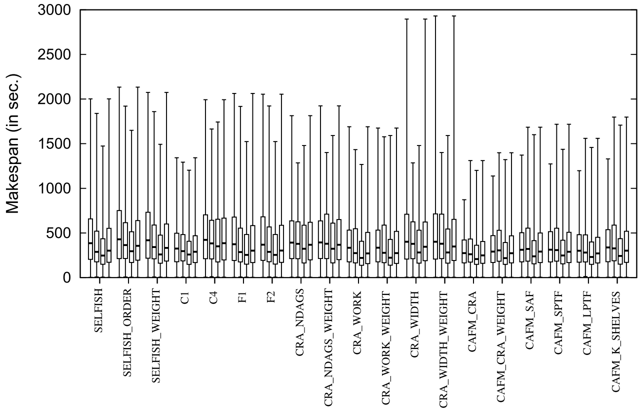


Fig. 3. Distribution of overall makespan values for all algorithms in box-and-whiskers fashion. For each algorithm, results are shown for random, Strassen, FFT, and all PTGs.

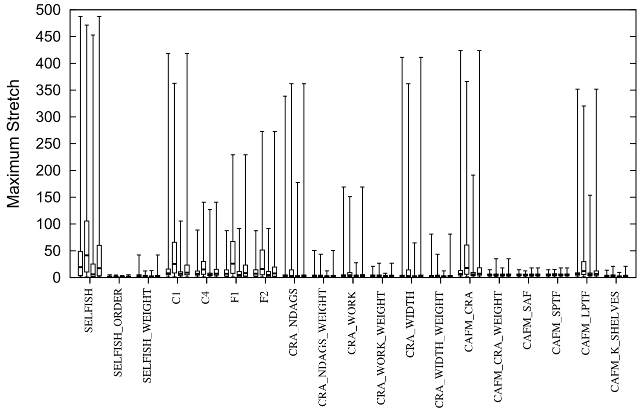


Fig. 4. Distribution of maximum stretch values for all algorithms, in box-and-whiskers fashion. For each algorithm, results are shown for random, Strassen, FFT, and all PTGs.

than if it were scheduled alone on the platform using the HCPA algorithm in [22]. This happens in cases with few concurrent PTGs and is explained as follows. HCPA bounds the number of processors that can be used in a schedule as a way to increase parallel efficiency. It turns out that this bound can be more stringent than the bound used by the CRA algorithms, and thus can lead to longer schedules.

5.2.2. Makespan distribution

Results for the makespan metric are shown in Fig. 3, which is similar to Fig. 2. With respect to the sensitivity of our results to the PTG populations, we find that the relative difference between the median value for two PTG populations ranges between 17.0% (for C4) and 39.8% (for CAFM_CRA_WEIGHT), and is 29.1% on average. Like in the previous section, we evaluate the sensitivity of the algorithm ranking to the PTG populations. For the random, FFT, and Strassen PTG populations we find that the average ranking differences with the ranking computed considered all PTGs are 2.0, 2.4, and 1.15. Here again, we conclude that using results computed over all PTGs is reasonable for analyzing our results, which is what we do in the rest of this discussion. However, we note two notable exceptions: the maximum makespan achieved by CRA_WIDTH and CRA_WIDTH_WEIGHT is larger by roughly a factor 2 for the random PTGs than for the FFT and Strassen PTGs.

Using SELFISH as a reference, the only algorithms that outperform it noticeably for one or more quartile statistics are C1, CRA_WORK, CRA_WORK_WEIGHT, and all the CAFM algorithms. Expectedly, the ORDER and WEIGHT variants do not lead to improvements in makespan since for this metric the order in which the PTGs complete is of no consequence. The good performance of C1 is expected since it treats the PTGs as a single PTG and uses a scheduling heuristic that was designed to minimize the overall makespan. The best algorithm overall is CAFM_CRA, with the lowest maximum (1310.69), third quartile (404.73), median (249.35), and minimum (3.71), but its performance is comparable to that of C1, and also to that of the CRA_WORK, CRA_WORK_WEIGHT, and CAFM algorithms, which all have maximum makespans under 1800. Like in the previous section, we find that several algorithms perform reasonably well and that the makespan metric does not designate a single clear winner. For instance, CAFM_K_SHELVES, which is found in the previous section to be among the two best algorithms in terms of scaled sum completion time, has median value only 20% larger than that of CAFM_CRA.

5.2.3. Maximum stretch distribution

Fig. 4 shows results for the maximum stretch metric. These results are even more consistent across PTG populations than those in the previous sections with respect to median values: the average ranking differences between rankings for random, FFT, and Strassen PTGS and the ranking computed over all PTGs are 0.84, 0.84, and 1.89, respectively. The biggest difference is 6, for F1 and F2. In the discussion hereafter we use statistics computed over all PTGs. However, we see that for the maximum and third quartile statistics some algorithms are sensitive to the PTG populations. This is seen as variation of the position of the top whisker within each group of the box-and-whisker diagram, which is particularly noticeable for C1 and CRA_WIDTH. Regardless, unlike the previous two metrics, the maximum stretch is discriminating as several algorithms are significantly outperformed by a few others. These better algorithms have results that are consistent across PTG populations.

Using the maximum stretch as a metric for fairness (low maximum stretch indicating good fairness), the least fair algorithm is SELFISH, with a median value of 17.49 and a maximum value of 487.69. SELFISH_WEIGHT performs roughly an order of magnitude better (median value of 2.38 and maximum value of 42.18), but is not as good as SELFISH_ORDER (median value of 1.79 and maximum value of 5.23). This is consistent with results in the two previous sections. We see that all algorithms based on those of Zhao et al. in [31] achieve larger maximum values than SELFISH_ORDER by at least an order of magnitude. Among the algorithms in the CRA family, the best algorithm is CRA_WORK_WEIGHT. In the CAFM family, CAFM_SAF, CAFM_SPTF, and CAFM_K_SHELVES are the best algorithms. An expected observation is that the algorithms that achieve the best worst-case fairness (i.e., the ones with the lowest top whiskers in the figure) are the ones that explicitly schedule shorter PTGs first. This explains the high maximum values (above 100) of SELFISH, CRA_NDAGS, CRA_WIDTH, CAFM_CRA, and CAFM_LPTF, which either do not pay special attention to short PTGs or actually schedule them later explicitly. One interesting feature of the results is that all CRA algorithms use the same mapping procedure, while all the CAFM algorithms use the same allocation procedure. In both families of algorithms we see some algorithms that perform very poorly and some algorithms that perform very well. For the two other metrics, all algorithms perform somewhat similarly, but for minimizing the maximum stretch (i.e., for avoiding cases in which a few PTGs experience very poor performance), both allocation and mapping must be done judiciously.

The fairest algorithm overall is SELFISH_ORDER, with a median value of 1.79, a third quartile of 2.40, and a maximum of 5.23. CAFM_K_SHELVES is strictly better than CRA_WORK_WEIGHT according to all quartile statistics, with a median value of 2.43, a third quartile of 4.18 and a maximum of 21.09. CAFM_SAF and

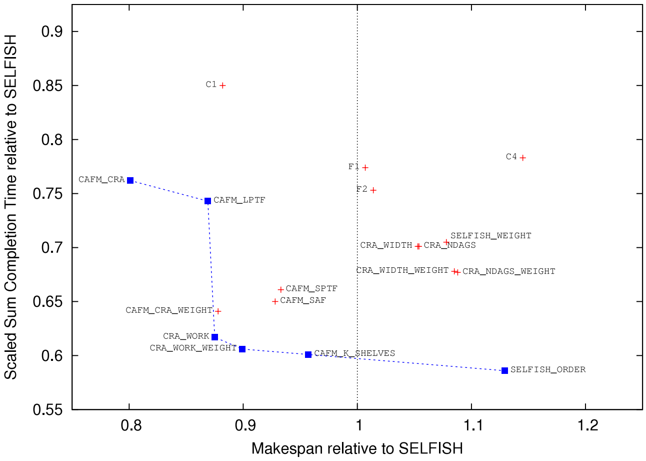


Fig. 5. Mean scaled sum completion time versus mean makespan, relative to SELFISH.

CAFM_SPTF achieve slightly better maximum values (17.98 in both cases), but have median values above 4.50, i.e., above the median value of CAFM_K_SHELVES. We conclude that the best algorithm in terms of maximum stretch is SELFISH_ORDER, followed by CAFM_K_SHELVES.

According to all results seen so far, one algorithm that is among the best for each metric is CAFM_K_SHELVES. SELFISH_ORDER is a close contender, and it outperforms CAFM_K_SHELVES for the maximum stretch metric.

5.2.4. Bi-criteria performance relative to *SELFISH*

The results obtained so far make it difficult to fully understand the trade-offs between the different metrics. In this section, we conduct a bi-criteria analysis of the algorithms, using SELFISH as a baseline so as to quantify how the different algorithms improve the schedules relative to the performance of the simplest algorithm. For each algorithm, each simulation scenario, and each performance metric, we compute a relative metric by dividing by the value achieved by SELFISH. Values larger (resp., lower) than 1 thus mean that an algorithm is less (resp., more) effective than SELFISH. For each algorithm, we then compute the harmonic mean of these ratios over all simulation scenarios. We do not use the arithmetic mean because SELFISH experiences high variations in our metrics across our experimental scenarios, meaning that a few ratios could dominate the arithmetic mean. The harmonic mean is typically used precisely to avoid such bias.

Fig. 5 shows all algorithms in a two-dimensional "Pareto" plot, where the *x*-axis is the mean relative makespan and the *y*-axis is the mean relative makespan. Algorithms located toward the bottom-left corner of the graph are preferable, and algorithms that are non-dominated, i.e., not outperformed by any single other algorithm for both the mean relative makespan and the mean relative scaled sum completion time, are linked with a line.

This figure confirms some of the findings in the previous three sections. We see that several algorithm can improve on the straightforward SELFISH approach for both criteria. The algorithms based on those of Zhao et al. in [31] are all significantly outperformed by other algorithms. We also see that the WEIGHT version of each algorithm lies to the right and below the original algorithm, meaning that it improves the scaled sum completion time but degrades the makespan. The same observation holds for SELFISH_WEIGHT and SELFISH_ORDER. Among the non-dominated algorithms, SELFISH_ORDER achieves the best scaled sum completion time, but leads to a makespan more than 10% larger than SELFISH. This demonstrates that improving upon SELFISH in terms of fairness is straightforward, but doing so while maintaining, or improving, the makespan requires more sophisticated algorithms. At the other end of the spectrum is CAFM_CRA, which achieves a makespan 20% lower than that of SELFISH. CAFM_LPTF also achieves a low makespan, but only slightly lower than that of CRA_WORK at the expense of a large increase in scaled sum completion time, as seen by the steep line segment connecting the two algorithms. CRA_WORK, CRA_WORK_WEIGHT, and CAFM_K_SHELVES are non-dominated, and form a spectrum of trade-offs between makespan and scaled sum completion time. Note that CAFM_CRA_WEIGHT is almost non-dominated, but, like CAFM_LPTF, does not seem to be a good alternative to CRA_WORK. Among the six non-dominated algorithms, CAFM_LPTF and SELFISH_ORDER both pay for a small decrease in one criterion by a large increase in the other.

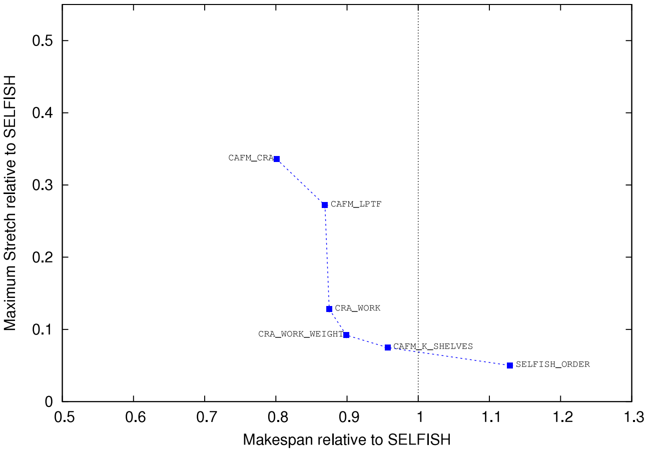


Fig. 6. Mean maximum stretch versus mean makespan, relative to SELFISH.

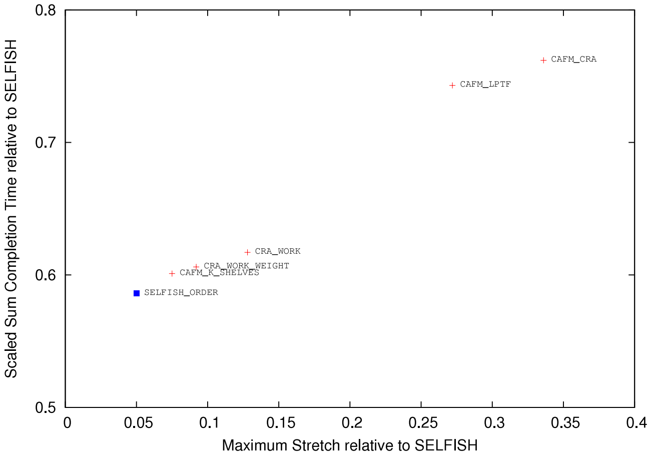


Fig. 7. Mean maximum stretch versus mean scaled sum completion time, relative to SELFISH.

Fig. 6 is similar to Fig. 5, but its *y*-axis shows the mean maximum stretch relative to SELFISH. This figure only shows the six non-dominated algorithms in Fig. 5. It turns out that most other algorithms are dominated for these two metrics as well, the exception being CAFM_CRA_WEIGHT. The observations made on Fig. 5 regarding the relative performance of the algorithms hold for Fig. 6. As in the previous sections, we see that some algorithms outperform SELFISH in terms of the maximum stretch by more than one order of magnitude.

Fig. 7 shows the mean maximum stretch versus mean scaled sum completion time, relative to SELFISH, for the six non-dominated algorithms shown in the previous figure. SELFISH_ORDER is the only non-dominated algorithm. It is therefore never dominated regardless of which two metrics are

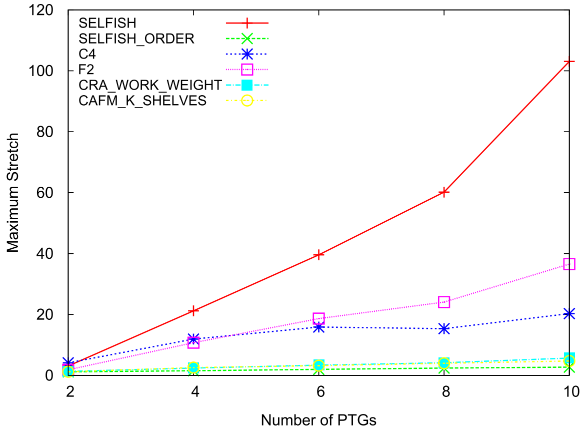


Fig. 8. Mean maximum stretch versus the number of concurrent PTGs.

considered. However, it leads to a very high makespan, more than 10% larger than that of SELFISH. Likely preferable in practice is CAFM_K_SHELVES, followed by CRA_WORK_WEIGHT.

5.2.5. Maximum stretch versus number of PTGs

In this section, we study how the maximum stretch, i.e., our measure of fairness, evolves as the number of concurrently scheduled PTGs increases. Presumably, as the number of PTGs increases, maintaining fairness among PTGs becomes more difficult. The results are shown in Fig. 8, which plots the mean maximum stretch, averaged over all simulation scenarios for a given number of scenarios, versus the number of PTGs. The baseline algorithm, SELFISH, shows a steep growth in mean maximum stretch as the number of PTGs increases. This is expected, since this algorithm pays no attention to fairness. To avoid unnecessary clutter, we present results only for the fairest algorithm in each algorithm family. We include two of the algorithms proposed by Zhao et al., C4 and F2, as they use significantly different approaches. These two algorithms lead to results that are better than those of SELFISH, with C4 being the better of the two. SELFISH_ORDER, CRA_WORK_WEIGHT, and CAFM_K_SHELVES have similar behavior, and exhibit only slowly increasing mean maximum stretch. We conclude that these algorithms are able to produce fair schedules even when the number of PTGs competing for resources increases.

5.2.6. Cluster utilization

We define the utilization of a cluster for an experimental scenario as the number of idle processors as a given instant, average over the overall makespan. Table 2 shows the utilization results for each algorithm, averaged over all experimental scenarios for a given cluster (see Table 1 for cluster specifications). All utilization levels reported in the table are above 75%, and 89.5% of them are 90% or higher. This indicates that, in our experiments, the PTGs can saturate the clusters. This confirms that we study high-load scenarios in which there is true competition for compute resources. The lowest utilization levels are achieved for the largest cluster (gdx), which is expected. Particularly notable are the utilization levels around 75% for that cluster when using the CAFM algorithms.

Table 2
Cluster utilization levels in percentage, averaged over all simulation scenarios for a given cluster and a given algorithm.

	chti	grillon	grelon	gdx
SELFISH	100.00	99.82	98.78	96.20
SELFISH_ORDER	100.00	99.77	97.72	94.30
SELFISH_WEIGHT	100.00	99.77	97.78	94.40
C1	99.87	99.44	95.85	89.47
C4	99.96	99.47	96.38	90.88
F1	100.00	99.82	98.68	96.18
F2	100.00	99.82	98.68	96.15
CRA_NDAGS	99.85	99.93	99.88	99.87
CRA_NDAGS_WEIGHT	99.33	98.146	99.02	98.80
CRA_WORK	99.96	99.92	99.87	99.91
CRA_WORK_WEIGHT	99.96	99.48	99.10	99.10
CRA_WIDTH	99.83	99.85	99.82	99.81
CRA_WIDTH_WEIGHT	99.24	98.14	98.81	98.74
CAFM_CRA	99.79	98.89	91.63	78.30
CAFM_CRA_WEIGHT	99.72	98.55	89.82	75.62
CAFM_SAF	99.77	98.64	90.34	76.76
CAFM_SPTF	99.77	98.63	90.26	76.67
CAFM_LPTF	99.66	98.50	90.05	76.05
CAFM_K_SHELVES	99.88	99.38	94.98	88.07

5.2.7. Comparison of scheduling times

In this section, we compare our algorithms in terms of time to compute a schedule. Times are measured on an Intel 2.20 GHz processor and for the most demanding scenarios in our experiments: scheduling 10 random PTGs on the largest cluster (216 processors). Cluster size has a high impact on the time to compute a schedule for several algorithms as it increases the convergence time of the allocation procedure of HCPA, which is used as a building block. All results hereafter are averages over five samples.

The baseline algorithm, SELFISH, computes schedules in 0.29 s. The CRA algorithms are faster, at 0.09 s. These algorithms first determine a resource constraint for each PTG. They then build the schedule of each PTG according to this constraint and thus on fewer processors, which takes less time than for the whole cluster as in SELFISH. F1 and F2 compute schedules in 0.20 s. C1 and C4 compute schedules in 2.22 s. These algorithms schedule a very large composite graph for which the bottom levels of unscheduled tasks must be recomputed at each iteration. The CAFM algorithms all take significantly more time, around 50 s. 85% of this time is due to determining the specification of each PTG as a moldable job, which invokes HCPA many times. The performance of HCPA is very sensitive to the shape of the PTGs, being particularly poor for PTGs that have low task parallelism. Note that if a PTG is executed more than once, then its specification as a moldable job does not need to be recomputed. In this case the CAFM algorithms would exhibit runtimes under 15 s, still orders of magnitude larger than the runtimes of the CRA algorithms.

6. Conclusion

In this paper, we have evaluated algorithms for off-line scheduling of multiple PTGs on a cluster. We have used three metrics to quantify the quality of a schedule, namely, the scaled sum completion time, the makespan, and the maximum stretch. The first two metrics pertain to performance and the third to performance and fairness. Based on our experiments, three out of the 19 studied algorithms have emerged as possible contenders for consideration. We have found that CAFM_K_SHELVES is the best algorithm, considering its performance on all three metrics. This algorithm computes resource allocations based on the guaranteed algorithm in [12] for optimizing the makespan, and builds the schedule as a sequence of “shelves” of increasing durations based on the makespan guarantee. Another algorithm that achieves performance close to that of CAFM_K_SHELVES is CRA_WORK_WEIGHT. This algorithm is based on one of the algorithms in [21], and uses a much simpler allocation and mapping procedure. This algorithm may therefore be a good choice for large problem instances, for which the time needed by CAFM_K_SHELVES to compute the schedule may be prohibitively large. CRA_WORK_WEIGHT would also be a good choice for practitioners that prefer a less involved implementation of the scheduling algorithm. Finally, the even simpler SELFISH_ORDER outperforms both these algorithms in terms of maximum and scaled sum completion time, but performs poorly in terms of makespan. If makespan is not a metric under consideration, then SELFISH_ORDER is the algorithm of choice.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful suggestions that have greatly contributed to improving the quality of this work. This work has been partially supported by the ANR project SPADES (08-ANR-SEGI-025) and the ANR project GWENDIA (ANR-06-MDCA-009).

References

[1] K. Aida, H. Casanova, Scheduling mixed-parallel applications with advance reservations, Cluster Computing Journal 12 (2) (2009) 205–220.

[2] S. Bansal, P. Kumar, K. Singh, An improved two-step algorithm for task and data parallel scheduling in distributed memory machines, Parallel Computing 32 (10) (2006) 759–774.

[3] M. Bender, S. Chakrabarti, S. Muthukrishnan, Flow and stretch metrics for scheduling continuous job streams, in: Proceedings of the ACM/SIAM Symposium on Discrete Algorithms, SODA, January 1998, pp. 270–279.

[4] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, O. Richard, A batch scheduler with high level components, in: Proceedings of the 5th International Symposium on Cluster Computing and the Grid, CCGrid, May 2005, pp. 776–783.

[5] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, M. Melab, G. Mornet, R. Namyst, P. Primet, O. Richard, E.-G. Talbi, I. Touche, Grid'5000: A large scale and highly reconfigurable experimental grid testbed, International Journal of High Performance Computing and Applications 20 (4) (2006) 481–494.

[6] H. Casanova, F. Desprez, F. Suter, From heterogeneous task scheduling to heterogeneous mixed parallel scheduling, in: Proceedings of the 10th International Euro-Par Conference, in: Lecture Notes in Computer Science, vol. 3149, Springer-Verlag, August 2004, pp. 230–237.

[7] H. Casanova, A. Legrand, M. Quinson, SimGrid: a generic framework for large-scale distributed experiments, in: Proceedings of the 10th Int. Conference on Computer Modeling and Simulation, March 2008.

[8] S. Chakrabarti, J. Demmel, K. Yelick, Modeling the benefits of mixed data and task parallelism, in: Proceedings of the 7th Symposium on Parallel Algorithms and Architectures, SPAA, 1995, pp. 74–83.

[9] H. Chen, M. Maheswaran, Distributed dynamic scheduling of composite tasks on grid systems, in: Proceedings of the 12th Heterogeneous Computing Workshop, HCW, 2002.

[10] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press/McGraw-Hill, 1990.

[11] P.-F. Dutot, L. Eyraud, G. Mounié, D. Trystram, Bi-criteria algorithm for scheduling jobs on cluster platforms, in: Proceedings of the 16th ACM symposium on Parallelism in Algorithms and Architectures, SPAA, 2004, pp. 125–132.

[12] P.-F. Dutot, G. Mounié, D. Trystram, Scheduling parallel tasks—approximation algorithms, in: J.Y.-T. Leung (Ed.), Handbook of Scheduling, CRC Press, 2004 (Chapter 26).

[13] P.-F. Dutot, T. N'takpé, F. Suter, H. Casanova, Scheduling parallel task graphs on (almost) homogeneous multi-cluster platforms, IEEE Transactions on Parallel and Distributed Systems 20 (7) (2009) 940–952.

[14] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, Parallel job scheduling—a status report, in: Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 3277, Springer-Verlag, 2004.

[15] Grid5000. https://www.grid5000.fr.

[16] M.A. Iverson, F. Özgüner, Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment, Distributed System Engineering 1 (3) (1999).

[17] C.B. Lee, A. Snavely, Precise and realistic utility functions for user-centric performance analysis of schedulers, in: Proceedings of the IEEE International Symposium on High-Performance Distributed Computing, HPDC, June 2007.

[18] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Transactions on Parallel and Distributed Computing 12 (2001) 529–543.

[19] T. N'takpé, F. Suter, Critical path and area based scheduling of parallel task graphs on heterogeneous platforms, in: Proceedings of the 12th International Conference on Parallel and Distributed Systems, ICPADS, 2006, pp. 3–10.

[20] T. N'takpé, F. Suter, Self-constrained resource allocation for parallel task graph scheduling on shared computing grids, in: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS, November 2007.

[21] T. N'takpé, F. Suter, Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations, in: Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC, May 2009.

[22] T. N'takpé, F. Suter, H. Casanova, A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms, in: Proceedings of the 6th International Symposium on Parallel and Distributed Computing, ISPDC, July 2007, pp. 250–257.

[23] S. Ostermann, P. Radu, T. Fahringer, A. Iosup, D. Epema, Trace-based characteristics of grid workflows, in: T. Priol, M. Vanneschi (Eds.), From Grids to Service and Pervasive Computing, in: CoreGRID, vol. 10, Springer-Verlag, 2008, pp. 191–204.

[24] A. Radulescu, A. van Gemund, A low-cost approach towards mixed task and data parallel scheduling, in: Proceedings of the 15th International Conference on Parallel Processing, ICPP, September 2001.

[25] R. Sakellariou, H. Zhao, A hybrid heuristic for dag scheduling on heterogeneous systems, in: Proceedings of the 13th Heterogeneous Computing Workshop, HCW, April 2004.

[26] U. Schwiegelshohn, R. Yahyapour, Fairness in parallel job scheduling, Journal of Scheduling 3 (5) (2000) 297–320.

[27] The SimGrid project. http://simgrid.gforge.inria.fr.

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

[29] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.

[30] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, J. Saltz, An integrated approach for processor allocation and scheduling of mixed-parallel applications, in: Proceedings of the 35th International Conference on Parallel Processing, ICPP, August 2006, pp. 443–450.

[31] H. Zhao, R. Sakellariou, Scheduling multiple DAGs onto heterogeneous systems, in: Proceedings of the 15th Heterogeneous Computing Workshop, HCW, April 2006.

Henri Casanova is an Associate Professor in the Information and Computer Sciences Department at the University of Hawai'i 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'Informatique et d'Hydraulique de Toulouse, France, in 1993, his M.S. from the Université Paul Sabatier, Toulouse, France, in 1994, and his Ph.D. from the University of Tennessee, Knoxville, US, in 1998.

Frédéric Desprez is a Senior researcher at INRIA and holds a position at LIP laboratory (ENS Lyon, France). He received his Ph.D. in Computer Science from the Institut National Polytechnique de Grenoble in 1994 and his MS in Computer Science from the Ecole Normale Supérieure de Lyon in 1990. He held a post-doctoral researcher position at ICL laboratory (The University of Tennessee, Knoxville) in 1994 and a position of assistant professor at ENSEIRB/LaBRI in 1995. His research interests include scheduling for large-scale distributed platforms, SOA architectures, data management, and grid computing. He is managing the ERCIM Working Group CoreGrid on Grids, P2P and Service Computing, which involves more than 31 institutions across Europe. He is also leading the Aladdin project, which manages the Grid'5000 experimental Grid. Within the Decrypthon project (AFM, CNRS, IBM), he is responsible for the DIET middleware and the port of application over the Decrypthon platform. In 2008, he received a faculty award from IBM for his work around data distribution and scheduling for Grid and Cloud platforms.

He has authored 26 international journal papers and 57 international conference papers. See http://graal.ens-lyon.fr/~desprez/ for further information.

Frédéric Suter has been a CNRS junior researcher at the IN2P3 Computing 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.