Minimizing Stretch and Makespan of Multiple Parallel Task Graphs via Malleable Allocations

Henri Casanova[1], Frédéric Desprez[2], Frédéric Suter[3]

[1]*Department of Information and Computer Sciences, University of Hawai'i at Manoa, USA.*
henric@hawaii.edu

[2]*LIP UMR 5668, ENS Lyon, INRIA, CNRS, UCBL, University of Lyon, France*
Frederic.Desprez@inria.fr

[3]*IN2P3 Computing Center, CNRS/IN2P3, Lyon-Villeurbanne, France*
Frederic.Suter@cc.in2p3.fr

***Abstract*—Many scientific applications can be structured as Parallel Task Graphs (PTGs), i.e., graphs of data-parallel tasks. Adding data-parallelism to a task-parallel application provides opportunities for higher performance and scalability, but poses scheduling challenges. We study the off-line scheduling of multiple PTGs on a single, homogeneous cluster. The objective is to optimize performance and fairness. We propose a novel algorithm that first computes perfectly fair PTG completion times assuming that each PTG is an ideal malleable job. These completion times are then relaxed so that the schedule is organized as a sequence of periods and is still close to the perfectly fair schedule. Finally, since PTGs are not perfectly malleable, the algorithm increases the execution time of all PTGs uniformly until it can successfully schedule each task in a period. Our evaluation in simulation, using both synthetic and real-world application configurations, shows that our algorithm outperforms previously proposed algorithms when considering two different performance metrics and one fairness metric.**

I. Introduction

Scientific applications executing on parallel computing platforms can exploit two types of parallelism: *task parallelism* and *data parallelism*. A way to achieve higher scalability and performance is to design parallel applications with 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 and 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*. Task execution time is typically non-decreasing as the allocation increases. PTGs arise naturally in many applications (see [1] 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 in 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, or running fewer concurrent tasks each with a larger allocation?

Commodity clusters are popular platforms that are typically shared between several applications. In this work we consider the sharing of a cluster among multiple PTGs in an off-line context: several PTGs have been submitted by different users and are ready to execute at the same time. The objective is twofold: (i) to minimize the performance of all the PTGs; and (ii) to maximize the fairness among the PTGs.

Three approaches have been proposed in the literature that relate to the above problem. In [2], Zhao et al. propose several 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 [3], 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 statically according to various criteria pertaining to the characteristics of the PTGs. Finally, in [4], Dutot et al. propose bi-criteria algorithms for scheduling independent moldable jobs, based on an approximation algorithm for optimizing the

makespan. Enhanced algorithms derived from these approaches have been described and extensively evaluated in [5]. Therein, it is shown that the best algorithms, that is the one that achieve good tradeoffs between performance and fairness, all rely on a same principle: Each PTG is scheduled on a subset of the processors in the cluster, and these subsets remain fixed throughout the execution of the whole batch of PTGs.

In this paper we investigate whether better schedules can be obtained by relaxing the constraint that each PTG executes on a fixed subset of the available processors. Our rationale is that, to increase fairness, it should be beneficial to allocate more processors to short PTGs so that they can complete earlier. These processors can then be later redistributed among other PTGs. The allocation of each PTG is thus *malleable*, i.e., decomposed in several periods with a potentially different number of allocated processors in each period. Our main contribution is a novel algorithm, Malleable Allocations with Guaranteed Stretch (`MAGS`), that determines such periods and computes allocations within these periods. The periods and allocations are based on a relaxation of a perfectly fair schedule assuming that PTGs are ideal malleable jobs. This relaxation comes with a guarantee on a metric that relates to both performance and fairness. Our evaluation in simulation, using both synthetic and real-world application configurations, shows that our algorithms leads to better schedules than the best algorithms in [5].

This paper is organized as follows. Section II details our platform and application models, and gives a precise problem statement. Section III reviews the different approaches proposed in the literature. Section IV describes the `MAGS` algorithm, which we evaluate experimentally in Section V. Section VI concludes the paper with a summary of our findings and future work directions.

II. Problem Statement

A. Platform and Application Models

A cluster consists of P *processors* interconnected by a high-speed network. A PTG is modeled as a 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*, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges representing precedence constraints between tasks.

While we model task precedence constraints, we do not model data transfers between tasks. Our rationale is that in the generated schedules a task may complete well before the beginning of one or more of its successors. This affords 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. The overhead of storing output data and of loading input data is comprised in each task's execution time as an overhead that depends on 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. We also refer to the allocation of a task v as $p(v)$. 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.

B. Metrics and Problem Statement

We consider the simultaneous execution of N PTGs on a cluster. The problem is to allocate processors to the tasks of these PTGs and to schedule them. We consider three metrics to quantify the quality of a schedule.

A popular metric to evaluate the level of performance achieved by a job that competes with other jobs is the *stretch* [6]. The stretch of a PTG is defined as the makespan achieved in the presence of resource contention divided by the makespan that would have been achieved if the PTG had had dedicated use of the cluster. For instance, if a PTG could have run in 2 hours using the entire cluster, but instead ran in 6 hours due to competition with other PTGs, then its stretch is 3. A lower stretch value denotes better performance.

For PTG $i = 1, \ldots, N$, we use $C^*_{max_i}$ to denote the makespan on the dedicated cluster, and C_{max_i} to denote the makespan in the presence of competition with the other PTGs. We quantify the overall performance of the PTGs using the *average stretch*, defined as follows:

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

Note that this definition is not the arithmetic mean of the

stretches. The arithmetic mean could seem more natural, but it is less stable than the above metric [5].

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

Finally, we need a metric to quantify the fairness of a schedule. A perfectly fair schedule is one in which all PTGs have the same stretch. There are several applicable definitions of fairness given in the literature. 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. Yet another possibility, which we adopt in this work, is to quantify fairness 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 [6]. Note, however, that we have analyzed our results with all three fairness definitions above and that the ranking of the algorithms studied in this paper is not sensitive to the chosen definition.

III. Related Work

In [5] four different approaches are proposed for scheduling a batch of PTGs on a cluster in a view to optimizing both performance and fairness. All presented 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, insuring that those processors are available to execute the task and that task precedence constraints are respected. These algorithms are compared to a naïve algorithm called `SELFISH`, which operates as follows. For each PTG, `SELFISH` uses the HCPA algorithm in [7] 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. Once allocations have been computed for all tasks of all PTGs, these tasks are scheduled on the cluster using a classical list-scheduling algorithm. In the produced schedule there may be opportunities to reduce fragmentation by moving task start times earlier. `SELFISH`, like all algorithms in [5], takes advantage of such opportunities with a post-processing phase inspired by the "conservative back-filling" technique used by batch schedulers [8]. We present hereafter the four algorithmic approaches in [5] and highlight the best performing algorithms in each approach.

A. `SELFISH` with Improved Task Mapping

One clear weakness of `SELFISH` is that it does not differentiate between "short" and "long" PTGs. It schedules all tasks of all PTGs together in order of decreasing bottom-levels, i.e., the distance of a task to the end of the PTG. The completion of a short PTG could be postponed, leading to a high stretch. `SELFISH_ORDER` is similar to `SELFISH`, but sorts tasks first by increasing $C^*_{max_i}$ values, and then by decreasing bottom-levels. This simply amounts to scheduling short PTGs before long PTGs.

B. Composing PTGs

In [2] 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. These algorithms were adapted in [5] to the case of PTGs. However, they were were consistently outperformed by those using the approaches described in the two next sections. Consequently, we do not include them in our evaluations.

C. Static Resource Constraints

In [3], N'takpé et al. propose to construct fair schedules for multiple PTGs by constraining the number of processors allocated to each PTG. More precisely, they ensure that no more than a fraction of the cluster is allocated to the tasks in a given precedence level of each PTG. The rationale is that ready tasks often belong to the same precedence level of a single PTG and thus can be executed concurrently. The authors consider several strategies to determine a static resource constraint for each PTG. Among them, the `CRA_WORK` strategy accounts for the amount of computation for the PTGs by specifying the resource constraint of the i^{th} PTG, P_i, as

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

where ω_i denotes the sequential work of the i^{th} PTG (i.e., the sum of the sequential execution times of all its tasks). The `CRA_WORK_WEIGHT` optimization proposed in [5] is similar to `CRA_WORK` but divides the bottom level of each task of the i^{th} PTG by the square of $C^*_{max_i}$ before the mapping phase. This heuristic attempts to weigh the bottom level of a task by the makespan of its

PTG so as to prioritize tasks belonging to short PTGs. CRA_WORK_WEIGHT was found in [5] to be the best algorithm for the static resource constraints approach.

D. Coarse-Grain Allocation, Fine-Grain Mapping

In [4], Dutot et al. propose algorithms for scheduling moldable independent jobs on a cluster. All their algorithms rely on the $3/2 + \epsilon$ approximation algorithm in [9]. This algorithm computes an approximation of the optimal makespan, C^*_{max}, and an allocation for each moldable job (i.e., a number of processors). 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. The goal is 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 optimization criteria used in [4]. Because its tasks can be executed on any number of processors, a PTG is a moldable job. The algorithm in [4] can thus be used to schedule multiple PTGs. Because designed for generic moldable jobs, the obtained schedule cannot take advantage of the fine-grain structure of PTGs, leading to schedule fragmentation.

The CAFM_K_SHELVES algorithm proposed in [5] is inspired by that in [4]. It first computes the execution time of each PTG assuming that p processors are available, with p varying from 1 to P. This is done with the HCPA algorithm in [7], and produces a specification of each PTG as a moldable job. The approximation algorithm in [9] is then used to compute an approximation of C^*_{max}, and the algorithm in [4] is used to obtain an allocation for each job in a K-shelf schedule. Each job is assigned a rectangular "box", which spans a number of processors and a number of time units. The individual tasks of the job, which is really a PTG, are then scheduled within this box.

E. Discussion

The results in [5] show that CAFM_K_SHELVES is the best among the aforementioned algorithms according to all three metrics. CRA_WORK_WEIGHT achieves performance close behind, with the advantage that it uses much simpler allocation and mapping procedures. This algorithm may therefore be a good choice for large problem instances, for which the time for CAFM_K_SHELVES to compute the schedule may be prohibitively large. Finally, the even simpler SELFISH_ORDER outperforms both these algorithms in terms of maximum and average stretch, but performs poorly in terms of makespan. We include all three algorithms as well as the baseline algorithm SELFISH in our evaluation in Section V.

IV. An Algorithm to Produce Malleable Allocations

The best algorithms in [5], CAFM_K_SHELVES and CRA_WORK_WEIGHT, both schedule each PTG within a rigid rectangular "box." In this work we develop an algorithm, MAGS, that structures the execution of the batch of PTGs as a sequence of time periods. Within each period each PTG is allocated a given number of processors, and each task of each PTG is scheduled within one of these periods. Consequently, a PTG can use different numbers of processors throughout its execution. In other words, allocations are *malleable*. We detail the steps of this algorithm hereafter, starting with the way in which periods are determined.

A. Determining the Scheduling Periods

To structure the schedule in periods we use a perfectly fair schedule as a starting point. In a perfectly fair schedule all PTGs experience the same stretch, S, and the best perfectly fair schedule is the one that leads to the smallest value for S. In this section we first compute a lower bound on S, called S^*. This lower bound is computed under the (unrealistic) assumption that each PTG is an ideal malleable job, i.e., composed of an infinite number of independent, infinitesimal tasks. It is therefore possible to allocate any number of processors to each job at any instant in time.

Recall from Section II that $C^*_{max_i}$ denotes the makespan of PTG i when scheduled on the dedicated cluster. Let us assume, without loss of generality, that $C^*_{max_i} \leq C^*_{max_{i+1}}$ for $i = 1, \ldots, N-1$. In a perfectly fair schedule that achieves a stretch S, job i completes exactly at time $S \times C^*_{max_i}$. We can now write simple constraints on the total work, i.e., execution time multiplied by number of allocated processors, of each job. For each i, all job j for $1 \leq j \leq i$ must be completed by time $S \times C^*_{max_i}$. Otherwise a PTG would have a stretch higher than S. Therefore, the sum of the works of each job j, for $1 \leq j \leq i$, must be lower than or equal to $P \times S \times C^*_{max_i}$. More formally:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} p \times C^*_{max_j} \leq p \times S \times C^*_{max_i} \ , \quad (1)$$

and we obtain the lower bound on the stretch as:

$$S^* = \max_{i=1,\ldots,n} \frac{1}{C^*_{max_i}} \sum_{j=1}^{i} C^*_{max_j} \ . \quad (2)$$

We wish to structure the schedule as a sequence of time periods. Within each period a job is allocated a number of processors. One possibility would be to have the period boundaries coincide with the job finish times. This would lead to N periods, with the boundaries at times $S^* \times C^*_{max_i}$, $i = 1, \ldots, N$. There could therefore be many periods (up to N), and, more importantly, some of these periods could be short when two jobs have similar $C^*_{max_i}$ values. Short periods are not a problem under the assumption that jobs are ideally malleable. However, in our schedule, each task of a PTG must be scheduled entirely within a period. We impose this constraint because, as we will see, it makes task allocation and task scheduling tractable. Unfortunately, this constraint also means that, in general, a short period may not be usable: all ready tasks of a PTG may simply have execution times larger than the period duration even for large allocations. Consequently, we propose a relaxation of the perfectly fair schedule so that we can reduce the number of periods and avoid short periods. This relaxation, described hereafter, comes with the guarantee that the maximum stretch over all PTGs is not more than a fixed factor away from the bound S^*.

We structure the schedule as M time periods. Period $i = 1, \ldots, M$ lasts from time t_{i-1} until time t_i, with $t_0 = 0$. t_i, $i = 1, \ldots, M$, and M are to be determined. We denote by i_j the index of the period during which job j completes in the perfectly fair schedule. More formally, $t_{i_j-1} \leq S^* \times C^*_{max_j} < t_{i_j}$. We set $t_1 = S^* \times C_{max_1}$ so that only job 1 may complete during the first period. We use geometrically increasing periods, relaxing the perfectly fair schedule so that each job $j > 1$ completes at time t_{i_j} rather than at time $S^* \times C^*_{max_j}$. Periods are defined geometrically by $t_{i+1} = t_i \times (1 + \lambda)$ for $i = 2, \ldots, M$ and some $\lambda > 0$. Therefore, $t_i = S^* \times C_{max_1} \times (1+\lambda)^{i-1}$ for $i = 1, \ldots, N$. Let use write the completion time of job $j > 1$ in the perfectly fair schedule as $t_{i_j-1} + \varepsilon$, for $\varepsilon \geq 0$. The stretch of job $j > 1$ in the relaxed schedule, S_j, is computed as:

$$
\begin{aligned}
S_j &= \frac{t_{i_j}}{C^*_{max_j}} \\
&= \frac{t_{i_j-1}(1+\lambda)}{C^*_{max_j}} \\
&= \frac{t_{i_j-1}(1+\lambda)}{\frac{1}{S^*}(t_{i_j-1}+\varepsilon)} \\
&= (1+\lambda)S^* \frac{t_{i_j-1}}{t_{i_j-1}+\varepsilon} \\
&\leq (1+\lambda)S^*
\end{aligned}
$$

We have therefore obtained a set of periods so that each job completes at the end of a period, with the guarantee that no job experiences a stretch higher than $S^*(1+\lambda)$.

Our algorithm uses $\lambda = 1$, meaning that the duration of period $i + 1$ is twice the duration of period i, and the maximum stretch is $2 \times S^*$. Note that our initial goal was to avoid short periods. We allow the specification of a bound on the smallest period, π, so that we do not generate any period of duration shorter than π. If $\pi > S^* C^*_{max_1}$, then we merge the first two periods into one of duration $S^* C^*_{max_1}(1 + \lambda)$, leading to $\lambda = \max(1, \pi/(S^* C^*_{max_1}) - 1)$. If instead $\pi \leq S^* C^*_{max_1}$, then the smallest period may be the second one, which is of length $S^* C^*_{max_1} \lambda$. We obtain $\lambda = \max(1, \pi/(S^* C^*_{max_1}))$. In summary, given the $C^*_{max_i}$ values and a specification of the smallest allowable period π, we can generate a set of periods for which, under the perfectly malleable parallel job assumption, the stretch of a job is guaranteed not to be more than a factor 2 away from S^*.

Once the periods are determined, our algorithm computes allocations within each period in the following way. At each period, available processors are allocated in a round-robin fashion to all jobs that must complete during the period. Jobs are then considered in order of increasing completion time. The algorithm allocates processors to each job greedily considering periods in order, until enough processors have been allocated to the job. Enough processors have been allocated once the sum of the works of the job at all periods (i.e., the number of allocated processors during a period multiplied by that period's duration) is equal to the job's total work.

Figure 1 illustrates this allocation scheme on an example. Six PTGs have to be scheduled concurrently on a cluster that comprises 47 processors (see Section V-A1 for a description of our target clusters). The values of $C^*_{max_i}$ are respectively 45, 76, 93, 102, 221 and 232 seconds. According to Equation 1, S^* is equal to $(45 + 76 + 93 + 102 + 221 + 232)/232 = 3.31$. This means that, in a perfectly fair schedule, the stretch of a job is at least 3.31.

Our algorithm generates periods ($[0; 190]$, $[190; 380]$, and $[380; 770]$). The smallest PTG ($C^*_{max_i} = 45$) is the only one to finish in the first period. Note that the two largest PTGs use no processors in the second period to let PTG 2, 3, and 4 complete earlier. Assuming perfectly malleable jobs, the respective stretches of the PTG in the relaxed schedule are at worse 4.2, 5.0, 4.08, 3.43, 3.48 and 3.31. Expectedly, these values are greater than S^* and lower that $2 \times S^*$.

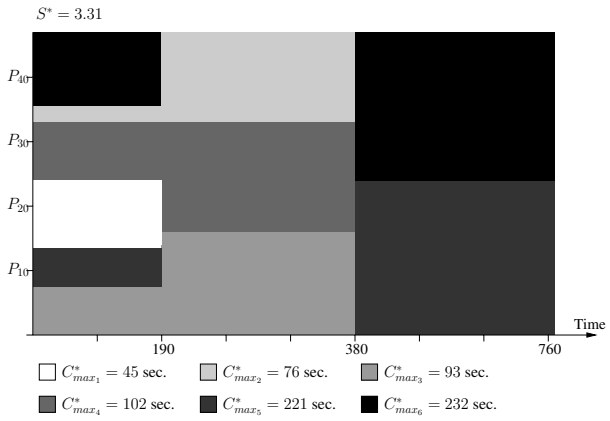

Figure 1. Example of period and malleable allocations for the concurrent scheduling of 6 PTGs on a cluster of 47 processors.

B. Scheduling PTGs in Malleable Allocations

Given allocations computed as described in the previous section, we must now schedule the tasks of the PTGs. Recall that the main objective is that PTG i completes no later than $S^* \times C^*_{max_i}$. In a view to achieving this objective, we consider the periods in reverse order and schedule the tasks of each PTG in a bottom-up fashion, i.e., starting from the exit tasks and moving towards the entry task. In this way, the exit task of each PTG finishes exactly at the end of the last period at which the PTG's allocation is non-zero. Implementing this scheme simply requires that a "reversed" copy of each PTG be created, which can be done easily by changing the direction of each edge $e_{i,j}$ in $\mathcal{E}$. The successors of task v_i become its predecessors and conversely. Finally, the entry and exit nodes exchange their roles. The question of how to determine the allocation and mapping of a task remains. For each period, each PTG is allowed to use a certain number of processors. To compute task allocations we use the HCPA algorithm [7], which is applied assuming a cluster that contains exactly this number of processors. We then select the task with the highest bottom level in the reversed PTG and map it on the set of processors that minimizes its finish time.

C. Relaxing the Stretch Guarantee

Depending on the periods and on the nature of the PTGs, it may not be possible to schedule all tasks. Indeed, the allocations are computed assuming that the jobs are ideally malleable. This is not the case in practice as PTG tasks are not infinitesimal. Therefore, some tasks may not be able to complete before the end of a period and may be postponed to the subsequent period. Tasks may then remain unscheduled after the the procedure described in the previous section finishes, meaning that the schedule is not feasible. If this situation arises, we introduce *slack* in the schedule, meaning that the guarantee on the maximum stretch is no longer $2 \times S^*$ but instead $slack \times 2 \times S^*$, where $slack \geq 1$. If $slack > 1$, then the schedule is still perfectly fair, but PTGs experience lower performance than when $slack = 1$. In such a schedule periods are longer and possibly more numerous. For a $slack$ value large enough, one is guaranteed to be able to compute a feasible schedule.

Our algorithm searches for the smallest $slack$ value that leads to a feasible schedule. It starts with $slack = 1$ and doubles it until a feasible schedule is found. A binary search is then used between this slack value and 1 to find the smallest $slack$ value that leads to a feasible schedule. Once such a schedule has been found, it is compacted using backfilling as done by all algorithms in [5].

D. The MAGS Algorithm

First we recall how HCPA determines an allocation for each task of a PTG on a dedicated cluster. Algorithm 1 presents this allocation procedure.

Algorithm 1 HCPA_allocate(PTG, P)

1: **for all** $v \in \mathcal{V}$ **do**
2: $\quad p(v) \leftarrow 1$
3: **end for**
4: **while** $T_{CP} > T_A$ **do**
5: $\quad v \leftarrow$ task $\in$ CP $\mid \left(\frac{T(v,p(v))}{p(v)} - \frac{T(v,p(v)+1)}{p(v)+1}\right)$ is maximum
6: $\quad p(v) \leftarrow p(v) + 1$
7: $\quad$ **Update** T_A **and** T_{CP}
8: **end while**

This allocation procedure starts by allocating one processor to each task. Then it increases some of these allocations to balance the length of the critical path, T_{CP}, and the average area, $T_A = \frac{1}{\min(P,\sqrt{V \times P})} \sum_i W(v_i)$. Each iteration of the procedure increases the allocation of the task belonging to the current critical path that will benefit the most of being allocated on one more processor. Finally the values of T_{CP} and T_A are updated.

Algorithms 2 and 3 summarize the steps of the MAGS algorithm that were detailed in the previous sections.

V. Experimental Evaluation

A. Experimental Methodology

We use simulation to compare MAGS, CAFM_K_SHELVES, CRA_WORK_WEIGHT, SELFISH_ORDER, and SELFISH. 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 v3.3.4 [10], [11] as the basis for our simulator. SimGrid provides the required fundamental

Algorithm 2 The MAGS algorithm

1: **for all** PTG i **do**
2: Compute $C^*_{max_i}$
3: **end for**
4: Sort the PTGs by increasing values of $C^*_{max_i}$
5: **for all** PTG i **do**
6: Make a reversed copy of PTG i
7: **end for**
8: $slack = 1$
9: **while** schedule(PTGs, $slack$) != ok **do**
10: $slack = slack \times 2$
11: **end while**
12: $maxslack = slack$
13: Binary search for the smallest $slack$ between 1 and $maxslack$ for which schedule() returns ok
14: Apply backfilling

Algorithm 3 schedule(PTGs, $slack$)

1: Determine periods and malleable allocations (see IV-A)
2: **for all** period **do**
3: **for all** reversed PTG i **do**
4: **for all** unscheduled task v of PTG i **do**
5: Determine t's allocation using HCPA
6: Schedule t if it fits in the current period
7: **end for**
8: **end for**
9: **end for**
10: **if** a least one task remains unscheduled **then**
11: **return** $!ok$
12: **else**
13: **return** ok
14: **end if**

abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms. We explain hereafter how we instantiate the models of Section II-A for the simulation experiments.

1) Platforms: We use configurations from four clusters in the Grid'5000 platform deployed in France [12], [13]. 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 ($100\mu s$ latency and $1Gb$ bandwidth). Table I summarizes the number of processors per cluster and the computation 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).

Cluster	grelon	grillon	chti	gdx
#proc.	120	47	20	216
Gflop/sec	3.185	3.379	4.311	3.388

Table I
CLUSTER CHARACTERISTICS.

2) Applications: To instantiate the PTG model described in Section II-A we need to define specific models for execution times of data-parallel tasks and for the structure of the task graph.

To model data-parallel task execution times, i.e., to model how $T(v, p)$ varies with p, we use a simple approach based on Amdahl's law. The model 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 I/O operations necessary for data communication between tasks. With this model, task execution time strictly decreases as the number of processors increases.

We consider 3 different classes of PTGs in our experiments. The first class includes random PTGs of different size, shape, and communication to communication ratio. We generate $1,296$ distinct PTGs of this kind. We refer the reader to our graph generation program and its documentation for full details on the graph generation algorithm [14]. We generate our PTGs using on the same parameter values as those used in [5]. We also consider real PTGs from the Strassen matrix multiplication and from the Fast Fourier Transform (FFT) applications. Both are classical test cases for PTG scheduling algorithms. 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, respectively). All Strassen PTGs have 25 tasks. We generate 120 FFT PTGs and 100 Strassen PTGs.

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 4 different platforms, we have $25 \times 4 = 100$ instances of our scheduling problem for each class of PTGs, for a total of 300 instances. All these PTG sets can be found in [14]. This amounts to a total of $1,500$ problem instances, since we use 5 different values for N. Each algorithm is executed for each instance.

B. Simulation Results

We present results for all algorithms for the average stretch metric, the makespan metric, and the maximum

stretch metric. We also study the average performance of each algorithm relative to MAGS, and the time need to produce a schedule for all algorithms. Finally we discuss the behavior of the $slack$ parameter used by MAGS.

1) Average Stretch Distribution: Figure 2 shows the distribution of average stretch 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 1st (25%) and 3rd (75%) quartiles, while the whiskers show the minimal and maximal values. The horizontal line within the box indicates the median, or 2nd 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.

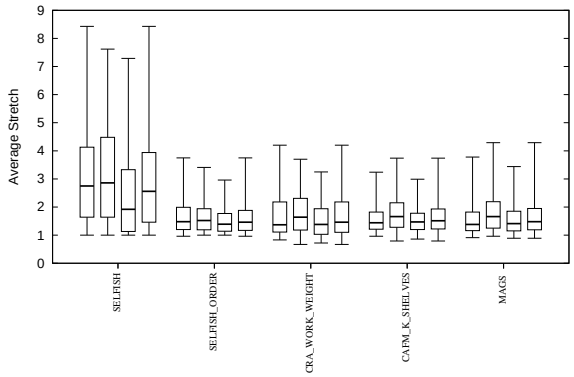

Figure 2. Distribution of average stretch values for all algorithms in box-and-whiskers fashion. For each algorithm results are shown for random, Strassen, FFT, and all PTGs.

Focusing on median values, we can see that the ranking of the algorithms does not depend heavily on the PTG populations. Using the results computed over all PTGs thus 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.

Expectedly, SELFISH leads to the worst results, with maximum average stretches up to 8.43. The SELFISH_ORDER variant is the best performing algorithm for this metrics, with a maximum average stretch of 3.75, a 3rd quartile of 2.46, a 2nd quartile of 1.71, a 1st quartile of 1.17, and a minimum of 0.96. However all other algorithms, including MAGS, achieve reasonable performance with maximum average stretches around 4 and median average stretches under 2.

2) Makespan Distribution: Results for the makespan metric are shown in Figure 3. Here again we see that results are consistent across PTG populations in terms of median values. We only discuss the fourth box and whisker diagram for each algorithm. One important observation is that SELFISH_ORDER is outperformed by SELFISH. Recall that SELFISH_ORDER is identical to SELFISH but for the fact that it changes the order in which the PTGs complete. This modification can hardly help reduce the makespan. And in fact, it hurts it because it changes the order in which the tasks are considered for backfilling. Considering the tasks of short PTGs first leads to later start times for the longer PTG, which ultimately increases the overall makespan.

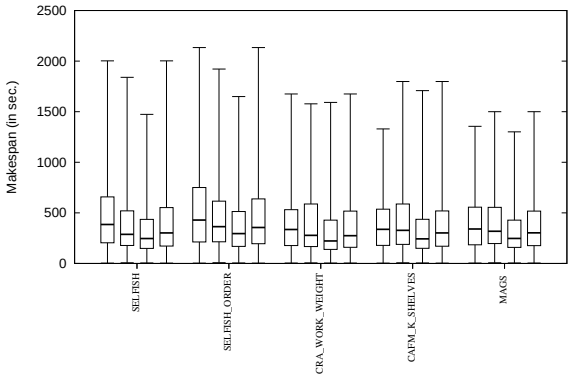

Figure 3. Distribution of overall makespan values for all algorithms in box-and-whiskers fashion. For each algorithm results are shown for random, Strassen, FFT, and all PTGs.

Considering quartile statistics for the other three algorithms, we see that they are either on par or better than SELFISH. For instance, all three are better in terms of maximum values. More specifically, CAFM_K_SHELVES and CRA_WORK_WEIGHT lead to better results for the first and second quartile (i.e., when the overall makespan of the batch of PTG is rather small). By contrast, MAGS leads to better improvements when the makespan is larger. For instance, MAGS reduces the makespan of the ten longest schedules by 30% on average against only 20% for CRA_WORK_WEIGHT.

3) Maximum Stretch Distribution: Figure 4 shows results for the maximum stretch metric. For sake of legibility, we do not display the upper part of the box-and-whisker diagrams of SELFISH. Across the complete PTG population, the top whisker value is 487.69 while the third quartile value is 60.18.

Most of these results are more consistent across PTG populations than those in the previous sections. However, we see that for the maximum and 3rd quartile statistics CAFM_K_SHELVES and CRA_WORK_WEIGHT are sensitive to PTG populations. This is seen as the variation of the position of the top whisker within each group of box-and-whisker diagram. Unlike the previous two metrics,

the maximum stretch is discriminating, and it turns out that these two algorithms and SELFISH are significantly outperformed by SELFISH_ORDER and MAGS. These better algorithms do have results that are consistent across PTG populations. Consequently, hereafter we use statistics computed over all PTGs.

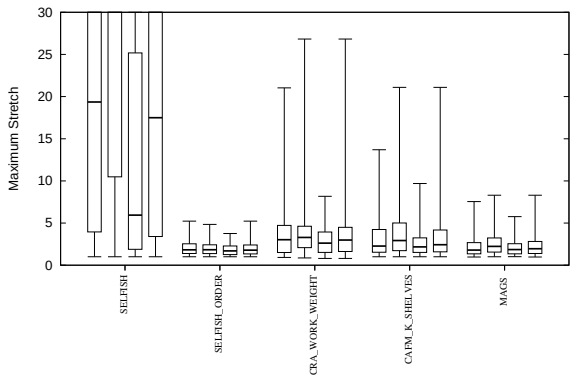

Figure 4. Distribution of maximum stretch values for all algorithms in box-and-whiskers fashion. For each algorithm results are shown for random, Strassen, FFT, and all PTGs.

Using the maximum stretch as a metric for fairness (low maximum stretch indicating good fairness), the least fair algorithm is SELFISH. The fairest algorithm overall is SELFISH_ORDER, with a median value of 1.79, a 3rd quartile of 2.40, and a maximum of 5.23. Then comes MAGS with a median value of 1.95, a 3rd quartile of 2.82, and a maximum of 8.30. The two remaining algorithms are competitive up to the 1st quartile. Their weakness is seen by from the top whisker value which is an order higher (21.09 for CAFM_K_SHELVES and 26.83 for CRA_WORK_WEIGHT).

In [5] CAFM_K_SHELVES was among the best algorithms for each metric. SELFISH_ORDER is a close contender and outperforms CAFM_K_SHELVES for the maximum stretch but leads to higher makespans. The results presented so far show that MAGS is competitive with these two algorithms for each metric. Two particular results are particularly noteworthy. MAGS outperforms SELFISH_ORDER in terms of overall makespan and outperforms CAFM_K_SHELVES in terms of maximum stretch.

C. Average Relative Performance of MAGS

In this section we provide insight on the performance of MAGS by computing relative improvements over its competitors. Results are shown in Table II for overall makespan, average stretch, and maximum stretch, averaged over all problem instances. A positive (resp. negative) value X in this table indicates that the average performance achieved by MAGS is $X\%$ better (resp. worse) than that of its competitor for the particular metric under consideration.

MAGS leads to better performance than SELFISH for all performance and fairness metrics. If the makespan improvement is marginal, those for the average and maximum stretch are large. This is expected as SELFISH lets the PTGs compete for the resources and may schedule longer PTGs first. SELFISH_ORDER is identified in [5] as an extremely fair algorithm. But such fairness is achieved by sacrificing the makespan. We see that MAGS outperforms SELFISH_ORDER significantly for the makespan (more than 20% better), while leading to maximum stretch that is within 14% of that of SELFISH_ORDER. We conclude that MAGS leads to a more reasonable compromise between performance and fairness than SELFISH_ORDER.

CAFM_K_SHELVES and CRA_WORK_WEIGHT are identified in [5] as the two best algorithms. MAGS leads to schedules that are significantly more fair than those produced by these two algorithms (by more than 38% and 49%, respectively). This observation was already made in the previous section. Our key result here is that this improvement in fairness is not at the expense of PTG performance, which was a weakness of SELFISH_ORDER. Indeed, on average MAGS is at most 2% worse than CAFM_K_SHELVES and CRA_WORK_WEIGHT for the makespan and the average stretch. And, in fact, on average it leads to a marginally better makespan than CAFM_K_SHELVES and marginally better average stretch than CRA_WORK_WEIGHT.

As expected the time needed to produce a schedule increases with the size of the cluster and the number and size of the PTGs to schedule. We have measured the time needed to produce a schedule for all algorithms, on an Intel 2.2GHz processor as an average over 5 problems instances. For each of these instances we schedule 10 random PTGs on the largest cluster. These times are 0.37, 0.23, 0.21, 197.1, and 0.86 seconds for SELFISH, SELFISH_ORDER, CRA_WORK_WEIGHT, CAFM_K_SHELVES, and MAGS, respectively. The large time for CAFM_K_SHELVES is because this algorithm computes the completion time of each PTG for all possible numbers of processors before building the schedule. Compared to the other three algorithms, MAGS takes between 2 and 4 times longer to produce a schedule. Regardless, its time to produce a schedule is reasonable since makespans are many orders of magnitude larger than a few seconds in practical settings.

	Makespan	Average Stretch	Maximum Stretch
SELFISH	7%	75.97%	1909.54%
SELFISH_ORDER	21.27%	−3.42%	−13.68%
CRA_WORK_WEIGHT	−1.99%	1.77%	49.79%
CAFM_K_SHELVES	1.14%	−0.44%	38.54%

Table II
AVERAGE PERFORMANCE OF CAFM_K_SHELVES, CRA_WORK_WEIGHT, SELFISH, AND SELFISH_ORDER RELATIVE TO MALLEABLE ALLOCATIONS WITH GUARANTEED STRETCH FOR THE THREE METRICS.

D. Guaranteed vs. Achieved Stretch

Recall that MAGS bases its schedule on a guarantee on the stretch computed assuming that PTGs are ideally malleable. The guarantee is that the stretch of each PTG is at most $2 \times S^*$. A $slack$ parameter is then used to account for the fact that PTGs are node ideally malleable in practice, making the guarantee $2 \times slack \times S^*$. We find that for 92% of our problem instances a feasible schedule is obtained with $slack = 1$. A value of $slack$ greater than 2 is necessary only for two instances. This shows that in the vast majority of the cases, the fact that the ideally malleable assumption is unrealistic does not have much impact. It was therefore a reasonable assumption to make to derive the overall structure of the schedule.

VI. CONCLUSION

In this paper, we have proposed a novel algorithm, MAGS, for the off-line scheduling of multiple PTGs on a cluster. This algorithm structures the schedule as a sequence of periods in time. Each PTG is given a (possible zero) allocation within each period. The periods and allocations are based on a relaxation of a perfectly fair schedule assuming that PTGs are ideal malleable jobs. This relaxation comes with a guarantee on the maximum stretch. Since the ideally malleable assumption does not hold in practice, the guarantee is further relaxed so as to accommodate the scheduling of all tasks of all PTGs. In the vast majority of our problem instances this additional relaxation was not needed. We have used three metrics to quantify the quality of a schedule, namely, the average stretch, the makespan, and the maximum stretch. The first two metrics pertain to performance and the third one pertains to performance and fairness.

In our experiments, when compared to a naïve algorithm that lets the PTGs compete for processors MAGS achieves better results for these three considered metrics. More importantly, MAGS achieves performance on par with that of previously published algorithms, but leads to a significant improvements of the fairness metric. Finally, MAGS produces schedules in a reasonable amount of time, which renders it usable in practice.

REFERENCES

[1] S. Chakrabarti, J. Demmel, and 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.

[2] H. Zhao and R. Sakellariou, "Scheduling Multiple DAGs onto Heterogeneous Systems," in *Proceedings of the 15th Heterogeneous Computing Workshop (HCW)*, Apr. 2006.

[3] T. N'takpé and 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.

[4] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram, "Bicriteria 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.

[5] H. Casanova, F. Desprez, and F. Suter, "On Cluster Resource Allocation for Multiple Parallel Task Graphs," Institut National de Recherche en Informatique et en Automatique, Tech. Rep. 7724, Mar 2010. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00461692/

[6] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and Stretch Metrics for Scheduling Continuous Job Streams," in *Proceedings of the ACM/SIAM Symposium on Discrete Algorithms*, 1998, pp. 270–279.

[7] T. N'takpé, F. Suter, and 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)*, Jul. 2007, pp. 250–257.

[8] A. Mu'alem and D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," *IEEE Transactions on Parallel and Distributed Computing*, vol. 12, pp. 529–543, 2001.

[9] P.-F. Dutot, G. Mounié, and D. Trystram, "Scheduling Parallel Tasks – Approximation Algorithms," in *Handbook of Scheduling*, J. Y.-T. Leung, Ed. CRC Press, 2004, ch. 26.

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

[11] "The SimGrid project," http://simgrid.gforge.inria.fr.

[12] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard, "Grid'5000: A large scale, reconfigurable, controlable and monitorable Grid platform," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, Nov. 2005, pp. 99–106.

[13] "Grid5000," https://www.grid5000.fr.

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