A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters

Frédéric Desprez, Frédéric Suter

A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters

Frédéric Desprez
INRIA, LIP, ENS Lyon
Lyon, France
Frederic.Desprez@inria.fr

Frédéric Suter
IN2P3 Computing Center, CNRS/IN2P3,
Lyon-Villeurbanne, France
Frederic.Suter@cc.in2p3.fr

***Abstract*—Applications structured as parallel task graphs exhibit both data and task parallelism, and arise in many domains. Scheduling these applications on parallel platforms has been a long-standing challenge. In the case of a single homogeneous cluster, most of the existing algorithms focus on the reduction of the application completion time (*makespan*). But in presence of resource managers such as batch schedulers and due to accentuated pressure on energy concerns, the produced schedules also have to be efficient in terms of resource usage. In this paper we propose a novel bi-criteria algorithm, called biCPA, able to optimize these two performance metrics either simultaneously or separately. Using simulation over a wide range of experimental scenarios, we find that biCPA leads to better results than previously published algorithms.**

I. Introduction

Scientific applications executed on parallel computing platforms can exploit two types of parallelism: *task parallelism* and *data parallelism*. A task-parallel application is partitioned into a set of tasks with possible precedence and communication constraints to form a *task graph*. A data-parallel application typically 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, and achieve higher scalability and performance, is to write parallel applications that use both types of parallelism, using what is often called *mixed parallelism*. With mixed parallelism applications are structured as *parallel task graphs* (PTGs), that is, task graphs of data-parallel tasks. PTGs arises naturally in many applications (see [1] for a discussion of the benefits of mixed parallelism and for application examples.)

One well-known challenge for executing PTGs is *scheduling*, that is, making decisions for mapping computations to platform resources to optimize 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*. A moldable task can be executed on various numbers of processors, with more processors hopefully leading to faster execution times. This raises the question of how many processors should be allocated to each data-parallel task. In other words, what is the best trade-off between running more concurrent data-parallel tasks each with fewer processors, or running fewer concurrent tasks each with more processors?

Typical platforms for executing PTG are homogeneous commodity clusters. The resources of such computing platforms are often accessed through a batch scheduler that is the most common resource management system used in production. A batch scheduler allocates, in a certain order, computing resources to the different requests submitted by users [2]. A known drawback of a batch scheduler is that users and resource owners have disconnected aims. Users usually want their applications to finish as soon as possible. A batch scheduler tries to ensure a maximal usage of the resources even if some particular requests have to be delayed in the process. In the particular context of PTG scheduling, two scenarios are possible. First, a user can build a *pre-schedule* of his/her mixed-parallel application. Then he/she submits a rigid version of it, i.e., in which all the processor sets allocated to each task have been fixed, to the batch scheduler. A second option is to let the scheduler determine these processor sets, provided that tasks can use different numbers of processors. Such a feature is for instance available in the OAR resource management system [3] and described in [4]. In this case the batch scheduler tries to use the resources as efficiently as possible. In other words it tries to minimize the work associated to the execution. Using less resources can also be one of the user's goals. A less resource consuming schedule will indeed be "greener". It can also lead to a lower bill if resource consumption is accounted.

In this paper we propose a new algorithm to schedule a Parallel Task Graph on a cluster. The performance objectives of this algorithm are: (i) to minimize the completion time of the PTG; and (ii) to minimize the amount of resources allocated for the schedule. We will show how the proposed algorithm is able to find a good trade-off between these two antagonistic objectives. We will also see that both metrics can be optimized separately to come up to either user or batch scheduler expectations.

Several algorithms have been proposed in the literature to schedule PTGs on clusters either in one [5] or two steps [6], [7], [8]. Most of these algorithms focus only on reducing the completion time of the scheduled applications and may lead to an inefficient use of the resources as pointed out in [9]. Among the two-step algorithms, the CPA (Critical Path and Area-based scheduling) algorithm in [8] was a pioneering work. Some drawbacks of this algorithm were the initial motivation of this work. We will show the limitations of previous improvements of this seminal algorithm. We will also demonstrate how the algorithm proposed in this paper produces schedules that outperform those of its competitors, both in terms of makespan and work reduction.

This paper is organized as follows. Section II discusses related work. Section III describes our application and cluster models, and gives a precise problem statement. Section IV details the proposed biCPA algorithm, which we evaluate experimentally in Section V. Finally Section VI summarizes our contribution and presents future work.

II. Related Work

Several algorithms have been recently proposed to schedule a single PTG on a cluster. Most of these algorithms decompose the scheduling in two phases. First they determine a resource *allocation* for each task of the PTG. Then they *map* the allocated tasks on the computing resources. Determining an allocation consists in fixing the number of processors to execute a moldable task. Previously published results show that, among the two-step algorithms, the CPA (Critical Path and Area-based scheduling) algorithm in [8] has low computational complexity and was shown to lead to good results when compared to its competitors. CPA aims at finding a good tradeoff between the length of the *critical path* and the *average area* (which measures the sum of the processor-time area required by the PTG tasks). A glaring drawback of CPA has been highlighted by [6] and [9]. Indeed for some application and platform configurations, CPA produces allocations that are too large and reduce concurrency in a way that is detrimental to performance.

The MCPA (Modified CPA) algorithm in [6] addresses this drawback by preventing the tasks belonging to a same level of precedence to be allocated on more processors than the cluster comprises. Nevertheless this improvement is limited to PTGs structured as a sequence of levels comprising independent tasks. For more irregular tasks graphs, MCPA is likely to build the same schedules as CPA.

A modified version of the HCPA (Heterogeneous CPA) algorithm in [9] uses a more stringent stopping criterion in allocation procedure. Stopping this allocation process earlier results in smaller allocations and thus increase the possible concurrency. This criterion was determined empirically and leads to a formulation of the average area that is no more homogeneous with the critical path length. Furthermore, HCPA is sometimes too conservative and stops the allocation process while some improvement is still possible.

On the other hand, the one-step iCASLB algorithm was shown to lead to better performance than some two-step algorithms, including CPA, at the price of a higher complexity [5]. This algorithm performs allocation and mapping simultaneously with a look-ahead mechanism to avoid being trapped in local minima and a backfilling approach to improve the schedule.

Finally there exist some theoretical results on the scheduling of moldable tasks with dependencies. In [10], the authors present a guaranteed two-step algorithm. Its allocation step relies on a relaxed linear program minimization and which also results in fractional processor allocations. A rounding procedure is then used to obtain integral allocations. The second step applies a simple list scheduling approach to map tasks. The guaranteed performance ratio is defined as the maximum ratio between the produced makespan and the optimal makespan. It is shown that the guaranteed performance ratio of this algorithm is $\sim$5.24 in the general case. This result was improved in [11], leading to a $\sim$4.73 performance ratio in the general case. The algorithm proposed in [10] was implemented and compared to HCPA in [12]. It was shown that non-guaranteed algorithms were competitive with the guaranteed one on the average but with tremendously shorter scheduling times.

III. Problem Statement

A. Platform and Application Models

A cluster consists of P compute nodes, or *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. Each processor is able to execute a certain amount of floating operations (or flop) per second that represents its computing speed. A processor can communicate with several other processors simultaneously under the *bounded multi-port model*. All the concurrent communication flows share the bandwidth of the communication link that connects this processor to the remaining of the cluster.

A PTG is modeled as a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \ldots, V\}$ is a set of vertices representing data-parallel tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of E edges representing precedence constraints between tasks. We distinguish two PTGs models that correspond to two different execution scenarios. These models differ on the weight assigned to the edges.

In the first model the edges are zero-weighted. This means that we do not model any communication network or any data transfer between tasks, with the following rationale. A schedule can be seen as a set of resource reservations, one per task, submitted to a batch scheduler. We may construct schedules in which a task may complete well before the beginning of one or more of its successors. Then it precludes the use of network communication between tasks. One solution is then to implement communication using disk I/O via files. The overhead of such communication is comprised in the task performance model described hereafter (as a sequential overhead).

Conversely the second PTG model considers that the execution takes place within a single reservation. It is then possible to resort to network communication for data transfers. Each edge $e_{i,j}$ is weighted by the amount of data (in bytes) that task v_i must send to task v_j. Note that in addition to data communication itself, there may be an overhead for data movements, e.g., when task v_i is executed on a different number of processors than task v_j.

Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task. Since data-parallel tasks can be executed on various numbers of processors, we denote by $T(v, p)$ 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 denote by $W(v) = T(v, p) \times p$ the work needed to execute the task v on p processors and by $BL(v)$ its bottom level, i.e., its distance in terms of execution time to the end of the application. 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. The makespan is denoted by C in the remaining of this work.

B. Metrics and Problem Statement

We consider the execution of a PTG on a cluster. The problem is to allocate resources to the tasks of this PTG and to schedule it so as to minimize its makespan and minimize the resource usage associated with this execution. These objectives are antagonistic and are respectively related to user and batch scheduler concerns. Indeed using more resources is likely to lead to smaller makespans while a lesser resource usage may increase the overall execution time. We thus have to solve a bi-criteria optimization problem.

In the following we define the makespan as $C = \max_i C(v_i)$ where $C(v_i)$ is the finish date of task v_i. To express the resource usage of a schedule, we denote by W the total work needed to execute the PTG. The definition of W depends on the model of PTG. In absence of network communications, it corresponds to the sum of the work needed to execute each allocated task, i.e., $W = \sum_i W(v_i)$. When the schedule is executed within a single resource reservation, it graphically corresponds to a box whose width is equal to the number of processors ("size" or $peakAlloc$) and height is equal to the makespan. The total work is then defined as the area of this box. The reservation's width corresponds to maximal number of processors simultaneously alloted in the schedule. We represent this value by $peakAlloc$. We then have $W = C \times peakAlloc$.

IV. A Bi-Criteria Scheduling Algorithm

In this section we first recall the principle of the allocation procedure of the CPA algorithm in [8]. We also explain how MCPA [6], and HCPA [9] algorithms modify this procedure. Then we detail the principle of the proposed *Bi-criteria CPA* (biCPA) algorithm.

A. Existing Allocation Procedures

Algorithm 1 *CPA allocation procedure

```
1: for all $v \in \mathcal{V}$ do
2:     $p(v) \leftarrow 1$
3: end for
4: while $T_{CP} > T_A$ do
5:     $v \leftarrow$ task $\in$ CP | $\left(\frac{T(v,p(v))}{p(v)} - \frac{T(v,p(v)+1)}{p(v)+1}\right)$ is maximum
6:         and $prec\_alloc(v) < P$ // for MCPA only
7:     $p(v) \leftarrow p(v) + 1$
8:     Update $T_A$ and $T_{CP}$
9: end while
```

Algorithm 1 presents the pseudo-code of the allocation procedure common to the CPA, MCPA and HCPA algorithms. This allocation procedure starts by allocating one processor to each task (lines 1-3). Then it increases some of these allocations to balance the length of the critical path, $T_{CP} = \max_i BL(v_i)$, and the average area, $T_A = \frac{1}{P} \sum_i W(v_i)$ (lines 4-9). As task mapping has not been computed yet, the critical path computation does not take into account the communications costs due to data dependencies.

Note that HCPA uses an other definition of T_A to stop this procedure before producing too large allocations. In HCPA, the average area is defined as $T_A = \frac{1}{\min(P, \sqrt{V \times P})} \sum_i W(v_i)$. 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 (lines 5-7). MCPA adds another condition to the selection of this task. It ensures that the sum of the allocations of the tasks in a same precedence level does not exceed the capacity of the cluster (line 6). Finally the values of T_{CP} and T_A are updated (line 8).

As stated in Section II, the allocation procedure of the CPA algorithm may reduce concurrency in a way that is detrimental to performance. This occurs when the number of processors in the cluster is much bigger than the number of tasks in the PTG. In such a configuration, the average area T_A grows very slowly due the division by P. This implies a large number of iterations to reach the tradeoff between the length of the critical path and the average area. This may lead to large allocations for some independent tasks that cannot be executed concurrently. The solution proposed in the HCPA algorithm allows the allocation procedure to converge faster. But it does not preserve the homogeneity of the $T_{CP} > T_A$ relation. Indeed T_{CP} represents a time while T_A is a ratio between a time and a number of tasks.

B. The biCPA Algorithm

Figure 1 shows how the critical path length T_{CP} and the average area T_A evolve during the allocation procedure of CPA. Note that it corresponds to the allocation of a PTG of 50 tasks on a cluster that comprises 20 processors. Then it is a case unfavorable to CPA with $V << P$. We can see that 96 iterations are needed to reach the desired tradeoff between T_{CP} and T_A.

The number of iterations of this allocation procedure strongly depends on how T_A and T_{CP} evolve. Due to the speedup model, the gain on T_{CP} tends to decrease as more processors are allocated. The slope of the evolution curve of T_A depends on the number of processors P. By taking the average over a value P' smaller than P, this slope will be steeper. The allocation procedure will then converge faster. For instance, if $P' = 10$ in Figure 1, the allocation procedure will stop after 34 iterations. Now we need to find the appropriate value of P'.

biCPA will base its new allocation procedure on a new definition of the average area denoted as T'_A. This variant of the average area is defined by

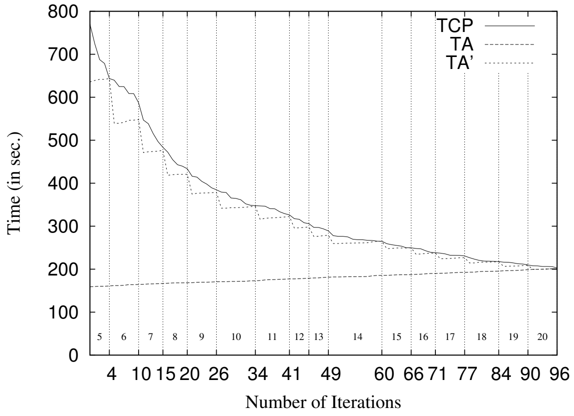

Figure 1. Evolution of T_{CP}, T_A, and T'_A throughout the allocation procedure of CPA for a random PTG of 50 tasks on a cluster of 20 processors.

$$T'_A = \frac{1}{P'} \sum_i W(v_i), \tag{1}$$

Figure 1 shows the evolution of T'_A throughout the allocation procedure. The value of P' is incremented each time T'_A becomes larger than T_{CP}. The evolution of P' is depicted by the labels on Figure 1.

An interesting thing is that each time P' is incremented the current task allocations correspond to those that would have been determined by CPA if the cluster has comprised P' processors. Moreover all these intermediate allocations can be determined during the execution of the original allocation procedure of CPA. These intermediate allocations are the key information needed by biCPA to find the best compromise between makespan and work.

Algorithm 2 presents the allocation procedure of biCPA which relies on the definition of T'_A. As explained before, the main difference with the allocation procedure of CPA lies in the most extern for loop (lines 4-14). This loop is used to set the value of T'_A that will be used in the inner loop (lines 6-10). Note that this inner loop actually corresponds to an interval of iterations of the seminal allocation procedure of CPA, as shown in Figure 1. Each time $T_{CP} \leq T'_A$, the current allocation is stored for each task (lines 11-13). At the end of this procedure, P different allocations are associated with each task in the PTG.

Then the second step of the biCPA algorithm consists in getting an estimation of the makespan and total work that can be achieved with each of these allocations. To obtain these performance indicators, the biCPA algorithm uses a classical list scheduling function. The task

Algorithm 2 The biCPA allocation procedure

```
for all $v \in \mathcal{V}$ do
    $p(v) \leftarrow 1$
end for
for $i = 1$ to $P$ do
    $T'_A = \frac{1}{i}\sum_j W(v_j)$
    while $T_{CP} > T'_A$ do
        $v \leftarrow$ task of the critical path | $\left(\frac{T(v,p(v))}{p(v)} - \frac{T(v,p(v)+1)}{p(v)+1}\right)$
        is maximum
        $p(v) \leftarrow p(v) + 1$
        Update $T'_A$ and $T_{CP}$
    end while
    for all $v \in \mathcal{V}$ do
        Store $p^i(v) \leftarrow p(v)$
    end for
end for
```

are first prioritized by decreasing bottom level. Then for each task, the set of processors that leads to the earliest finish time is selected. Once a mapping has been found for each task of the PTG, we determine the values of C_A and W_A. These value are stored in an array of structures. From these makespan and total work estimations, the biCPA algorithm is able to output four interesting schedules among all the schedules computed.

The first two schedules aims at optimizing one metric only. We can first select the allocation leading to the shortest estimation of the makespan. This allocation is found by sorting the array of structures by increasing makespan and picking out the first element. The second schedule produced by biCPA is the one that requires the smallest amount of work to execute the PTG. As for the first schedule, the corresponding allocation is found by sorting the array of structures, this time by increasing total work. Recall that our main aim is to design a bi-criteria scheduling algorithm. We then discard the solutions that leads to an improvement of the criterion to optimize but degrades the performance with regard to the other criterion, i.e., such that $C_i > C_P$ or $W_i > W_P$. Such a situation often occurs when trying to minimize the work needed to execute a PTG as only a few processors are used.

The process leading to the third and fourth schedules is more complex. The objective is now to optimize both metrics simultaneously. A first step is to determine, for each candidate allocation, the gain it offers with regard to each metric. This gain is measured by dividing the makespan and work achieved with the considered allocation respectively by the makespan and work obtained for P processors. A value less than one indicates a shorter completion time or less required work. Conversely, a ratio greater than one shows a performance degradation. Note that these relative makespan and work also tell us if the schedule produced by CPA can be improved. Then we can determine which allocations lead to non-dominated solutions. The best trade-off between our two objectives can be found among these allocations.

In a multi-objective optimization problem, there are at least two ways of defining what should be a good trade-off. A first definition is to find a solution that leads to the same improvement on each criterion. In our particular context this means an allocation that reduces the makespan and work with regard to the allocation of CPA in the same proportion formalized by Equation 2

$$\frac{C_i}{C_P} = \frac{W_i}{W_P}, \tag{2}$$

where C_P and W_P represent the makespan and work achieved when using all the processors (as in CPA). C_i and W_i correspond to the same quantities but when the allocations are determined for a cluster of only i processors. If several solutions satisfy to Equation 2, the best one will be the one with the smallest relative makespan.

Another definition of a good trade-off is to maximize the sum of the improvements that a solution achieves on each criterion. In our context, small values for relative makespan or work mean better performance. The best trade-off will be the allocation that minimizes

$$\frac{C_i}{C_P} + \frac{W_i}{W_P}. \tag{3}$$

Ties produced by this equation are broken by selecting the one with the smallest work.

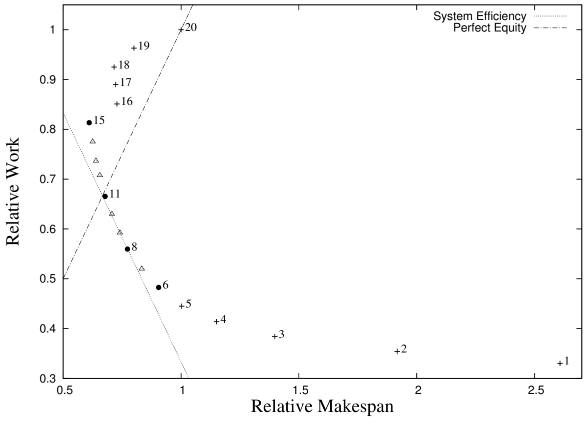

Figure 2. Evolution of C_i/C_P and W_i/W_P when i varies for a random PTG of 20 tasks without inter-task communications on a cluster of 20 processors.

Figure 2 illustrates and summarizes the different allocations selected by our biCPA algorithm. This figure shows the relative makespan (C_i/C_P) and work

(W_i/W_P) when i varies for the scheduling a random PTG of 20 tasks without inter-task communications on a cluster of 20 processors. The crosses depicts the discarded options, either because they are dominated (from 16 to 20) or degrading one of the criterion (from 1 to 5). The triangles correspond to the non dominated solutions while the black circles are the four values selected by our algorithm: (i) the best makespan improvement is achieved with $i = 15$; (ii) the best work without makespan degradation is obtained when $i = 6$; (iii) with $i = 11$, Equation 2 (whose solutions are shown by on the dashed line) is satisfied; and (iv) the sum of the two relative values is minimized when $i = 8$. The pointed line shows the system efficiency which corresponds to this minimal value.

In the remaining of this paper we distinguish the four schedules produced by our biCPA algorithm. biCPA-M is the schedule that minimizes the makespan while biCPA-W minimizes the work without degrading the makespan. The two schedules that optimizes both criteria are biCPA-E and biCPA-S. The former looks for the best solution to Equation 2 while the latter minimizes Equation 3.

C. Complexity Analysis

In this section we analyze the worst case complexity of the proposed biCPA algorithm. We first recall the complexity of the seminal CPA algorithm. MCPA and HCPA share the same complexity.

In the allocation step, the main loop (lines 4-9 in Algorithm 1) updates the bottom level of each task, to in turn update T_A. It also browses the critical path looking for the task to select and to update T_{CP}. In the worst case, both operations imply to consider all the nodes and edges of the PTG, e.g., for a chain graph, and take $\mathcal{O}(V+E)$ time. This loop can be executed $V \times P$ times as the allocation of each task can be incremented by one processor, starting from 1 and up to P. Note that this worst case can happen with a chain graph whose critical path is much larger than the area corresponding to a pure data-parallel execution of its tasks. However, the number of iterations needed by the allocation procedure to reach an equilibrium between T_{CP} and T_A is considerably less than $V \times P$ in more general cases. Consequently the worst case complexity of the allocation step of CPA is $\mathcal{O}(V\,P\,(V+E))$.

The mapping step of CPA can be divided in three components. It is first mandatory to set the priority of each task. As it corresponds to determining the bottom level in CPA, the associate complexity is $\mathcal{O}(V+E)$. Then the scheduling list is sorted according to these priorities in $\mathcal{O}(V\ \log V)$. Finally, it takes $\mathcal{O}(V \times P)$ time to schedule tasks on processors. This results in a total complexity of $\mathcal{O}(E + V\ \log V + V\,P)$. The overall complexity of the CPA like algorithms is then dominated by the allocation step and is $\mathcal{O}(V\,P\,(V+E))$.

As mentioned in Section IV, computing the P intermediary allocations does not increase the complexity of the allocation step of biCPA. However to select the allocation that optimizes the target metric, biCPA needs to build P different mappings. A factor of P is then added to the worst case complexity of the mapping step of CPA leading to $\mathcal{O}(P\,(E + V\ \log V + V\,P))$. The overall complexity of the biCPA algorithm is then $\mathcal{O}(V\,P\,(V+E+P))$. Thus it only adds $\mathcal{O}(VP^2)$ to CPA's complexity but it optimizes two criteria instead of one.

We finally recall the worst case complexity of the iCASLB algorithm in [5]. In the worst case of chain-like PTG, iCASLB takes $\mathcal{O}(V^3P^2 + VP^2E')$ time to build a schedule. E' corresponds to the number of edges in a modified version of the task graph. This modification adds an edge between two independent tasks v_i and v_j if v_j has to wait for the completion of v_i to start its execution due to resource constraints.

V. Experimental Evaluation

A. Experimental Methodology

We use simulation to compare and evaluate the algorithms. It allows us to perform a statistically significant number of experiments for a wide range of application and platform configurations (in a reasonable amount of time). We use the SimGrid v3.3 toolkit [13] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms. We present hereafter how we instantiate the models described in Section III-A for the simulation experiments and then present and discuss the results.

B. Platforms

We consider three clusters of the Grid'5000 platform [14]. One, named *chti*, is located in Lille, while the two others, named *grillon* and *grelon*, are located in Nancy. Each cluster uses a Gigabit switched interconnect internally ($100\mu sec$ latency and $1Gb/sec$ bandwidth). The *chti* cluster comprises 20 processors with a computing speed of 4.311 Gflop/sec, while the *grillon* cluster has 47 processors that computes 3.379 Gflop/sec. Finally the *grelon* cluster is made of 120 nodes, each of a computing speed of 3.185 Gflop/sec. These computing speed values were obtained with the HP-LinPACK benchmark over ACML.

C. Applications

To instantiate the PTG models described in Section III-A 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 1GByte of memory and thus $d \leq 121M$. We also assume that d is above $4M$. The volume of data communicated between two tasks is equal to $8 \times d$ bytes. We model the computational complexity of a task, in number of operations, with one of the three following expressions, which are representative of common applications: $a \cdot d$, $a \cdot d \log d$, and $d^{3/2}$. 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 complexities are chosen randomly among the three.

The above provides a model for sequential task execution, but we also need to model parallel executions, i.e., how $T(v, p)$ varies with p. We use a simple model based on Amdahl's law that is used extensively in the literature. It specifies that a fraction α of a task's sequential execution time is non-parallelizable. We simply pick random α values uniformly between 0% and 25%. With this "Amdahl model", an application task exhibits different execution times for different numbers of processors.

We consider random tasks graphs that consist of 20 or 50 data-parallel tasks. We use four popular parameters to define the shape of the PTG: width, regularity, density, and "jumps". The width determines the maximum parallelism in the PTG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the 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. Furthermore we add random "jumps edges" that go from level l to level $l + jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). The parameters have been chosen to cover a broad range of applications characteristics. They are not described into details due to the lack of space. We refer the reader to our DAG generation program and its documentation [15] for more details. This generator is voluntarily available to ease the reproduction of presented results. For each model, we generate 864 different PTGs.

While the above specifies a way to generate a population of synthetic PTGs, we also consider real PTGs from the Strassen matrix multiplication algorithm and from the Fast Fourier Transform (FFT) application. Both are classical test cases for PTG scheduling algorithms and we refer the reader for instance to [16] for details on their structure. These PTGs are more regular than our synthetic PTGs, which are more representative of workflow applications. The FFT PTGs have 2, 4, 8, or 16 levels (that is 5, 15, 39, or 95 tasks) while all the Strassen PTGs have the same number of tasks (25). For each model, we generate 400 different FFT task graphs and 100 Strassen PTGs.

D. Comparison of Scheduling Times

Before assessing the performance of the different algorithms, we compare these algorithms in terms of time to compute a schedule. This allows us to experimentally confirm the complexity study presented in Section IV-C. Times shown in Table I are measured on an Intel 2.20GHz processor and averaged over the whole range of simulation scenarios.

As our biCPA algorithm adds a factor P, the number of processors in the target cluster, to the complexity of the seminal CPA algorithm, we present this timing results on a per cluster basis.

	chti	**grillon**	**grelon**
CPA	0.01 sec. (0.03%)	0.11 sec. (0.22%)	1.49 sec. (3.04%)
HCPA	0.01 sec. (0.03%)	0.07 sec. (0.10%)	0.23 sec. (0.48%)
MCPA	0.01 sec. (0.03%)	0.06 sec. (0.14%)	0.85 sec. (1.92%)
biCPA	0.02 sec. (0.04%)	0.10 sec. (0.19%)	1.00 sec. (1.88%)
iCASLB	11.71 sec. (11.62%)	265.44 (365.4%)	N/A N/A

Table I
AVERAGE SCHEDULING TIMES AND PART OF THE EXECUTION TIMES OF THE DIFFERENT SCHEDULING ALGORITHMS DEPENDING ON THE TARGET CLUSTER.

We can see that the scheduling times of the algorithms derived from CPA, including biCPA are negligible with regard to the application execution time. Moreover the scheduling time of biCPA is very competitive in spite of its higher complexity. HCPA, and to a smaller extent MCPA, build schedule faster as their allocation procedure may stop earlier.

The last line of Table I shows the scheduling time of the algorithm with the highest worst case complexity, iCASLB. These times are orders of magnitude higher than those of its competitors. They even exceed the time needed to execute the scheduled application on a cluster of 47 processors. It prevented us to test iCASLB on the largest cluster. Nevertheless, these bad results mainly come from the most unfavorable type of PTG, i.e., chain-like task graphs. For such PTGs, iCASLB requires a large number of iterations to converge. If we do not count these PTGs in the average, the scheduling time of iCASLB becomes more competitive. It then takes 0.78 sec. and 8.83 sec. on average to schedule the remaining PTGs on the chti and grillon clusters respectively. iCASLB still requires up to 31% of the execution time to schedule our set of PTGs. In the remaining of our study, we compare the other algorithms to iCASLB only for the chti and grillon cluster.

E. Simulation Results

Figure 3 shows the distribution of the makespan across the whole range of simulation scenarios for each algorithm. 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 value (second quartile), while the cross indicates the average value.

The first observation is that the CPA algorithm is outperformed by all the other competitors for both types of PTGs (worst mean, median and maximum values). On the other hand the biCPA-M variant is the best performing. It is closely followed by the two bi-criteria variants, i.e., biCPA-S and biCPA-E. This shows the efficiency our optimization approach. The previously published modifications of CPA have a lesser impact, HCPA being better than MCPA. Finally biCPA-W is the worst performing of our proposed heuristics, but still better than CPA. This was expected as this variant only tries to not degrade the makespan while minimizing the work. An interesting thing to see is that biCPA-M reduces the maximum value by 25% with regard to CPA.

We now study the performance of the algorithm with regard to our second optimization objective. Figure 4 is similar to Figure 3 but shows the work distribution.

Almost the same comments as for the makespan can be made for the CPA, HCPA, MCPA, biCPA-S, and biCPA-E. But the relative performance of biCPA-M and biCPA-W is exactly the opposite. biCPA-M which was the best algorithm is now one of the worse, while biCPA-W is now the best performing algorithm. Such an inversion is actually obvious. Each variant performs the best with regard to the metric it aims at optimizing.

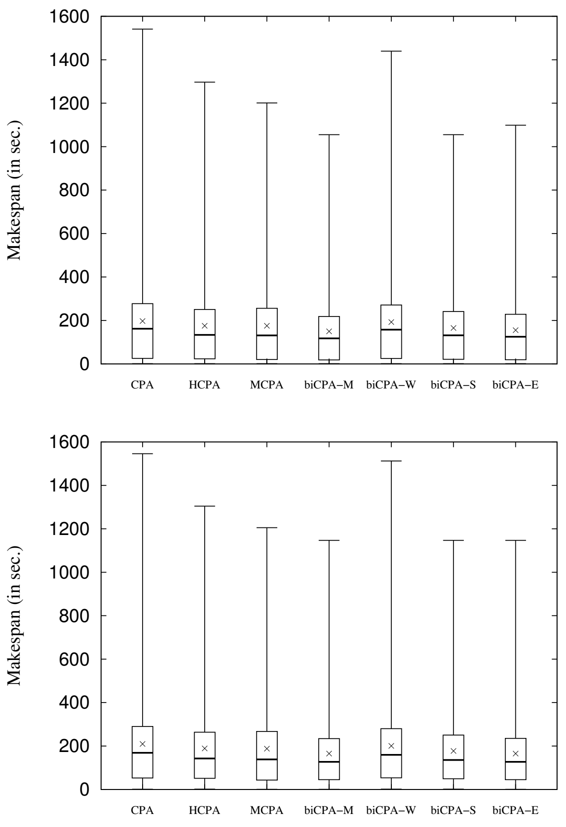

Figure 3. Distribution of overall makespan values for all algorithms for PTGs without (top) and with (bottom) inter-task communications.

Results so far have highlighted the conflict between makespan maximization and work minimization, which is often seen in multi-criteria optimization problems. In this section we present results that provide some insight regarding which algorithms achieve the best trade-offs between makespan and work. To this end, we use the CPA algorithm as a baseline comparator. For each simulation scenario and for each algorithm we compute the achieved makespan and work, relative to those achieved by CPA. We then compute these relative values averaged over all simulation scenarios. Figure 5 shows the relative performance of each algorithm with regard to CPA. The x-axis is the average relative makespan and the y-axis is the average relative work. It allows us to evaluate the gain with regard to contradictory metrics and to highlight

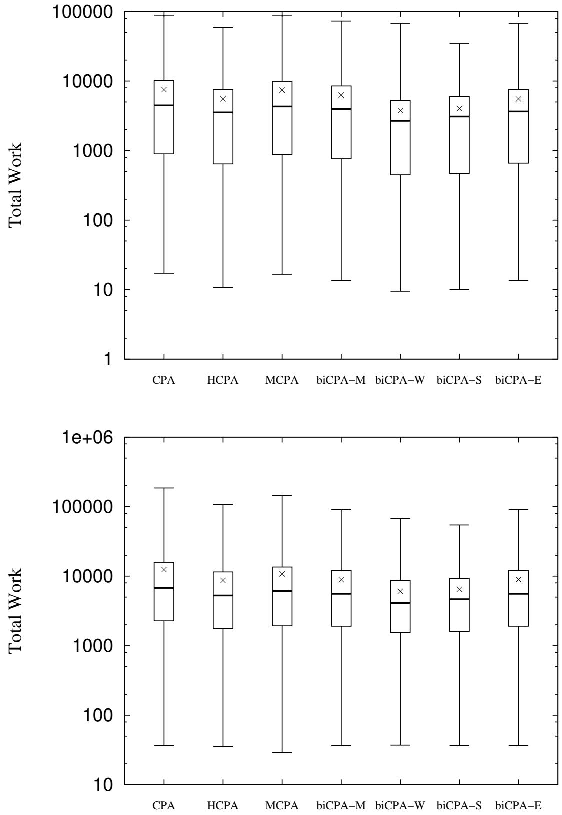

Figure 4. Distribution of overall work values for all algorithms for PTGs without (top) and with (bottom) inter-task communications.

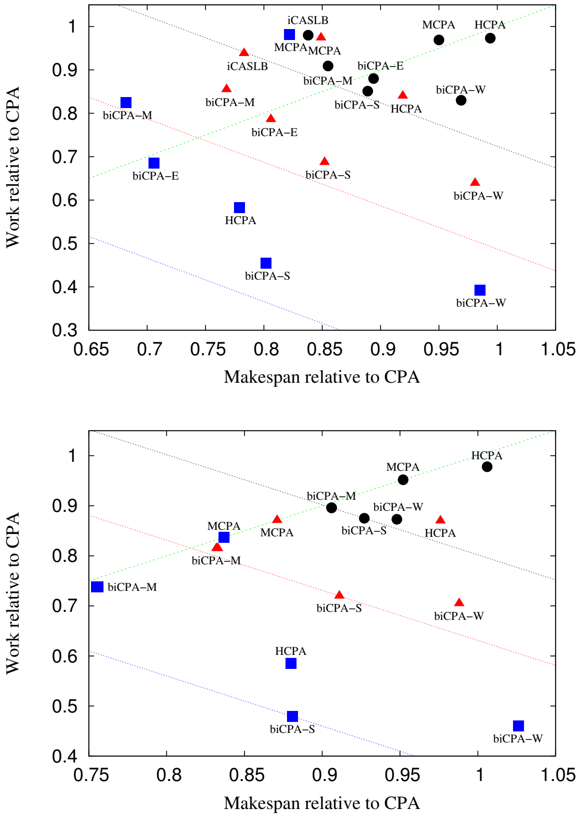

Figure 5. Bi-criteria average performance of all algorithms relatively to the CPA algorithm depending on the target cluster – small (chti – ●), medium (grillon – ▲) and large (grelon – ■) – for PTGs without (top) and with (bottom) inter-task communications.

dominant algorithms. Algorithms located toward the bottom-left corner of the graph are thus preferable.

As the quality of the schedules produced by CPA is supposed to decrease as the size of the cluster increases, we plot three distinct data sets. Performance on the chti cluster is represented by ●, that achieved on the grillon cluster by ▲ while ■ represent performance on the grelon cluster. For each target cluster, a pointed line shows the system efficiency. We also plot of perfect equity on this figure (dashed line). Note that in Figure 5 (bottom), biCPA-E is not plotted as it behaves exactly as biCPA-M. Indeed for PTG without communications, makespan and work are proportional. Hence a reduction of the makespan leads to an equivalent gain on the work, and satisfies Equation 2.

Figure 5 shows that all biCPA variants do improve CPA for both criteria. The only exception is for biCPA-W on the largest cluster with PTG with inter-task communication. In this particular case, the average makespan suffers an increase of 2.6% with regard to CPA. Despite our restriction to allocations that do not degrade one metric while optimizing the other, the communications negatively impact the makespan. Moreover our proposed heuristics outperform their competitors. Even iCASLB, whose complexity is higher is outclassed. HCPA, whose allocation stopping criterion is heterogeneous, is the only competitive one. HCPA actually becomes interesting when the number of processors always exceeds the number of tasks in the PTG. For instance on the smallest cluster, HCPA produces the same schedules as CPA. Our proposed biCPA algorithms are more generic and always lead to significant gains over CPA.

We also see that the gain offered by biCPA increases with the cluster size. This does not come from a degradation of CPA but from a constant improvement of our heuristics. While CPA reduces the average completion time of the PTG set by 10% when going from the small to the medium cluster, biCPA-M decreases it by 19%. The same occurs when going from the medium to the large cluster.

VI. Conclusion and Future Work

Applications represented by parallel task graphs exhibit a lot of parallelism. Several algorithms have been proposed for the scheduling of such applications on commodity clusters over the last years. All these algorithms focus on the reduction of the completion time of the application and assume dedicated access to the whole cluster. But the vast majority of the clusters in production are accessible through resource managers whose objectives are contradictory to that of users. Indeed if users aims at reducing their application execution time, batch schedulers are more interested in preventing the waste of resources. A common objective though, driven by energy saving concerns, is to reduce the resource usage associated with the execution of the application.

In this paper we proposed the biCPA bi-criteria scheduling algorithm that aims at optimizing two performance metrics: makespan and work. This algorithm produces four schedules that reach different trade-offs at the cost of an affordable complexity increase. All of them improve CPA and also outperform previously published optimizations of CPA. The performance of the four variants of the biCPA algorithm also clearly show their target audience. biCPA-M is typically a user oriented scheduling heuristic. On the other hand biCPA-W should be preferred by batch scheduler as it leads to almost the same completion time as CPA but with a drastic reduction of the resource usage. biCPA-E offers the fairest trade-off between both user's and resource manager's requirements. But the biCPA-S variant is the best candidate for a production use. Indeed it leads to a resource usage reduction similar to biCPA-W but with a higher benefit for the users. With regard to biCPA-E, it slightly favors the resource manager. On average biCPA-S reduces the makespan of 14% and the work of 34% compared to CPA.

As a future work we aim at transposing this work to the context of the scheduling workflows of multi-threaded routines on many-core architectures. This highly related scheduling problem is a very interesting and promising way to fully exploit the capacity of this emerging execution context. We would also like to assess the performance of our algorithm in a production context resorting to the OAR [3] resource manager or the workflow management system of the DIET middleware [17]. Finally more than two criteria could be considered. For instance adding energy concerns is an interesting topics. A common solution to deal with three criteria is to fix objectives for two of them and then try to optimize the third criterion.

VII. Acknowledgments

This work has been partially supported by the ANR project SPADES (08-ANR-SEGI-025).

References

[1] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the Benefits of Mixed Data and Task Parallelism," in *Proceedings of the 7th Symp. on Parallel Algorithms and Architectures (SPAA)*, 1995.

[2] D. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel Job Scheduling — a Status Report," in *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 3277, 2004, pp. 1–16.

[3] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A Batch Scheduler with High Level Components," in *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid)*, 2005.

[4] L. Eyraud, "A Pragmatic Analysis of Scheduling Environments on New Computing Platforms," *International Journal of High Performance Computing and Applications*, vol. 20, no. 4, pp. 507–516, 2006.

[5] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications," in *Proceedings of the 35th International Conference on Parallel Processing (ICPP)*, 2006.

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

[7] T. N'Takpé and F. Suter, "Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms," in *Proceedings of the 12th International Conferance on Parallel and Distributed Systems (ICPADS)*, 2006.

[8] A. Radulescu and A. van Gemund, "A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling," in *Proceedings of the 30th International Conference on Parallel Processing (ICPP)*, 2001.

[9] 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 Int. Symposium on Parallel and Distributed Computing (ISPDC)*, 2007.

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

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

[12] P.-F. Dutot, T. N'takpé, F. Suter, and H. Casanova, "Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 940–952, 2009.

[13] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSim'08)*, 2008.

[14] Grid'5000, "http://www.grid5000.fr."

[15] DAG Generation Program, http://www.loria.fr/∼suter/dags.html.

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

[17] A. Amar, R. Bolze, Y. Caniou, E. Caron, A. Chis, F. Desprez, B. Depardon, J.-S. Gay, G. Le Mahec, and D. Loureiro, "Tunable Scheduling in a GridRPC Framework," *Concurrency and Computation: Practice & Experience*, vol. 20, no. 9, pp. 1051–1069, 2008.