Scheduling Delta-Critical Tasks in Mixed-Parallel Applications on a National Grid

Frédéric Suter

▶ To cite this version:

Frédéric Suter. Scheduling Delta-Critical Tasks in Mixed-Parallel Applications on a National Grid. 8th IEEE/ACM International Conference on Grid Computing - Grid 2007, Sep 2007, Austin, TX, United States. pp.2-9. inria-00165868

HAL Id: inria-00165868
https://inria.hal.science/inria-00165868

Submitted on 18 Oct 2007

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

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

Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid

Frédéric Suter
Nancy Université / LORIA
UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1
Campus scientifique - BP 239
54506 Vandoeuvre-lès-Nancy, France
Frederic.Suter@loria.fr

***Abstract*— Mixed-parallel applications can take advantage of large-scale computing platforms but scheduling them efficiently on such platforms is challenging. When relying on classic list-scheduling algorithms, the issue of independent and selfish task allocation determination may arise. Indeed the allocation of the most critical task may lead to poor allocations for subsequent tasks. In this paper we propose a new mixed-parallel scheduling heuristic that takes into account that several tasks may have almost the same level of criticality during the allocation process. We then perform a comparison of this heuristic with other algorithms in simulation over a wide range of application and on platform conditions. We find that our heuristic achieves better performance in terms of schedule length, speedup and degradation from best.**

I. Introduction

The use of parallel computing for large and time-consuming scientific simulations has become mainstream. Two kinds of parallelism are typically exploited in scientific applications: *task parallelism* and *data parallelism*. In task parallelism, the application is partitioned into a set of tasks. These tasks are organized in a Directed Acyclic Graph (DAG) in which nodes correspond to tasks and edges correspond to precedence and/or data communication constraints. In data parallelism, an application exhibits parallelism typically at the level of loops, meaning that loop iterations can be executed, at least conceptually, in a Single Instruction Multiple Data (SIMD) fashion. A way to expose and exploit increased parallelism, to in turn achieve higher scalability and performance, is to write parallel applications that use both task and data parallelism. This approach is termed *mixed parallelism* and allows several data-parallel tasks to be executed concurrently. Mixed parallelism arises naturally in many applications and we refer the reader to [1] for application examples and a quantitative discussion of the benefits of mixed parallelism.

A well-known challenge for the efficient execution of task-parallel applications is scheduling. The problem consists in deciding which compute resource should perform which task when, in a view to optimizing some metric such as overall execution time. In the case of mixed-parallel applications, data parallelism adds a level of difficulty to the task-parallel scheduling problem. Indeed, the common assumption is that data-parallel tasks are moldable, i.e., they can be executed on arbitrary numbers of processors, with more processors leading to faster task execution times. This is typical of most mixed-parallel applications, and raises the question: how many processors should be allocated to each data-parallel task? There is thus an intriguing tension between running more concurrent data-parallel tasks with each fewer processors, or fewer concurrent data-parallel tasks with each more processors. Note that we assume that all tasks are known a priori, in an "off-line" fashion. Not surprisingly this scheduling problem is NP-complete (2-optimal algorithms are known) [2], [3], [4]. Consequently, several researchers have attempted to design scheduling heuristics for mixed-parallel applications. Most of the approaches proceed in two phases: one phase to determine how many processors should be allocated to each data-parallel task, the other phase to schedule these tasks on the platform using standard list scheduling algorithms.

Another approach, followed by the M-HEFT heuristic [5] consists in building a scheduling list sorted with regard to a given priority function and then determine an allocation for each task, one after the other, that minimizes a given objective function. A major issue may arise when considering tasks to schedule independently as the selfishly determined allocation of the first task may negatively impact the subsequent decisions for other tasks. In this paper we propose an original heuristic that consider tasks to schedule as groups of tasks having the same (or almost the same) priority. Depending on the size of such a group we bound the number of processors that can be allocated to a task to ensure that all the tasks of a same level of criticality can be scheduled concurrently. We perform empirical comparisons between this new heuristic, the original M-HEFT algorithm and two improved versions of M-HEFT [6] via extensive simulations for many application and real-world platforms. Our main finding is that our proposed heuristic leads to better performance with regard to several classical metrics.

This paper is organized as follows. Section II defines our application and platform models and states the scheduling problem. Section III reviews related work in detail and Section IV describes our original heuristic. Section V presents and discusses our experimental results. Finally, Section VI summarizes our findings and gives perspectives on future work.

II. Problem Statement

In this section, we review platform and application models that have been used in the literature for studying the scheduling of mixed-parallel applications onto heterogeneous platforms. We then discuss an important assumption and state the scheduling problem.

We consider a computing platform that consists of c clusters, where cluster $C_k, k = 1, \ldots, c$ contains p_k identical processors. A processor in cluster C_k computes at a speed s_k (in operations per seconds). Clusters may use different interconnect technologies, e.g., switches, among their processors. All clusters are interconnected together via a high-capacity backbone. Each cluster is connected to the backbone by a single network link. Inter-cluster communications are not serialized but happen concurrently, possibly causing contention on the network links between the clusters and the backbone.

A mixed-parallel application is modeled as a DAG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{t_i \,|\, i = 1, \ldots, N\}$ is a set of nodes representing data-parallel tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, N\} \times \{1, \ldots, N\}\}$ is a set of edges between nodes, representing communication between tasks. Each edge $e_{i,j}$ has a weight, which is the amount of data (in bytes) that task t_i must send to task t_j (we call t_j a *predecessor* of t_i). Note that in addition to data communication itself, there may be an overhead for data redistribution, e.g., when task t_i is executed on a different number of processors than task t_j. Since data-parallel tasks can be executed on various numbers of processors, we denote by $T^k(t,n)$ the execution time of task t if it were to be executed on n processors of cluster C_k. $T^k(t,n)$ accounts for both the computation and the communication costs involved when executing task t on cluster C_k. In practice, $T^k(t,n)$ can be measured via benchmarking on each cluster for several values of n or calculated via a performance model. The overall execution time, or *makespan*, is defined as the time between the beginning of the application's entry task and the completion of the application's exit task.

All previous work on mixed scheduling for heterogeneous platforms assumes that a data-parallel task must be executed on a homogeneous set of processors. In the above model this means that each data-parallel task must be executed within a single cluster since each cluster is homogeneous. This is reasonable because the latency of inter-cluster communications has a high impact on the performance of most data-parallel tasks and an intra-cluster execution is preferable. Furthermore, no good algorithm is known for data redistributions between data-parallel tasks running on sets of heterogeneous processors, and so it is not done often in practice.

Given a platform and an application we define the mixed parallelism scheduling problem as follows. For each task, determine the time at which it should start on which cluster and with how many processors, so that the overall execution time is minimized. The constraint is that data dependencies should be respected, i.e., a task cannot start before it has received all its input data from its predecessors. $N^k(t)$ denotes the number of processors allocated to task t scheduled on cluster C_k.

III. Previous Work

While a few authors have studied the scheduling of mixed-parallel applications from a theoretical perspective [2], [3], [4], several practical scheduling algorithms have been described in the literature [7], [8], [9]. Most of these algorithms proceed in two phases. In the first phase the algorithm computes the optimal number of processors for each data-parallel task of the application. In the second phase, the tasks are scheduled using one of the popular list scheduling algorithms. At any rate, previously obtained results show that the CPA algorithm described in [7] leads to the best schedules.

The fact that the allocation procedure of CPA produces large allocations that prevent the concurrent execution of independent tasks has been pointed out by MCPA [10] and HCPA [6] authors. Both algorithms propose a solution to stop the allocation procedure earlier and thus producing smaller allocations. MCPA enforces on the total number of processors allocated to critical tasks in the same level of precedence, *i.e.,* the "distance" from the beginning of the DAG, to be less than the number of processors of the platform. HCPA modifies the computation of the average processor utilization, which is part of the first phase's stopping criterion of CPA, to account both for the number of processors in the platform and for the number of tasks.

The M-HEFT algorithm [5] extends the HEFT [11] task scheduling algorithm to handle mixed parallelism. A glaring drawback of M-HEFT, as pointed out in [12], is that it tends to use very large processor allocations for application tasks. This is simply due to the fact that a task's processor allocation is chosen "blindly" so that the task's completion time is minimized. In [6] simple ways in which a task's processor allocation can be bounded are presented. It is also shown that when considering makespan and efficiency, which are the traditional metrics for evaluating the quality of a parallel execution, the simplest modification leads to results strictly superior to that achieved by M-HEFT. In this modified algorithm a task's allocation cannot use more than 50% of the processors of the cluster on which the task executes. On average, it reduces the makespan (by avoiding large allocations that could have a negative impact on the length of the critical path) but also improves efficiency (also by avoiding large allocations).

IV. Δ-Critical Tasks Scheduling

HEFT [11] is a list scheduling algorithm for scheduling a DAG of sequential tasks onto a heterogeneous set of processors. Recall that the bottom-level of a task is the length of the longest path from that task to the exit node. In the case of HEFT the length of a path is defined as the sum of the average computation time of each task and the average communication time of each communication edge along the path, where averages are computed over all processors and all network links. Tasks are scheduled in order of decreasing bottom-levels. Each task is scheduled using the allocation that minimizes its completion time, accounting for time spent in communication.

When extending this kind of algorithms to the case of data-parallel tasks on a platform that consists of heterogeneous clusters, the issue of determining on how many processors execute the task arises. As said in Section III a "blind" choice may lead to large processor allocations that can prevent the concurrent execution of some independent tasks. Consequently when considering a task (which is ready and with the highest bottom-level priority) for being scheduled, the scheduling algorithm has to take the other ready tasks into consideration, especially those having the same (or a close) bottom-level priority. Such tasks are indeed of the same level of criticality and the selfish, and potentially large, allocation of the first may delay the other tasks and could have a negative impact on the overall execution time of the application.

Depending on the structure of the application, the number of tasks having exactly the same bottom level priority varies. For instance, some parallel applications, *e.g.,* Strassen's matrix multiplication or one dimensional FFT algorithms, can easily be decomposed into regular precedence levels and tasks in a given level have the same bottom-level priority. When the decomposition into levels is more complex, *e.g.,* with execution path of different lengths or costs, the maximal number of tasks having *exactly* the same bottom-level priority is likely to be one. To efficiently schedule such application task graphs, we propose to relax the building of the set of the tasks to be considered as critical to include tasks having a bottom-level priority greater or equal to that of the most critical task minus a certain Δ. As all tasks are present in the scheduling list without distinction between ready or dependent tasks, a natural upper bound for Δ is the average execution time (as estimated in the computation of the bottom level) of the first task added into the set of Δ Critical Tasks (Δ-CT). Otherwise we may include tasks that can depend on the most critical task and thus cannot be executed concurrently. Consequently we first set Δ to that upper bound minus a certain ϵ. But a further investigation shown us that shorter schedules were produced by setting Δ to a smaller value that is the half of the average execution time of the first task added into a Δ-CT set. Our feeling is that including tasks with bottom-level priorities too close to those of the tasks depending on the most critical task has a negative impact on the remaining of the schedule. This leads to the Δ-CTS heuristic described by Algorithm 1.

Algorithm 1

```
Compute the bottom-level of each task t_i of the graph
Sort the tasks by decreasing bottom-level priority
while there are unscheduled tasks in the list do
  Build a Δ-CT from the first tasks of the list
  for all task t_i ∈ this Δ-CT do
    for all cluster C_j do
      Compute a_j the maximal allowed number of proc. on C_j
      Compute the best EFT(t_i, C_j, p_j) with 1 ≤ p_j ≤ a_j
    end for
    Assign t_i on the N^k(t_i) proc. of cluster C_k that minimize
    EFT(t_i)
  end for
end while
```

This algorithm first computes the bottom-level of each task of the DAG. To estimate it we simply compute averages over all possible 1-processor allocations over all clusters. Average communication times are computed over the set of communication times between all possible 1-processor allocations, not accounting for data redistribution costs but only for data transfer times. Then we sort the tasks in a scheduling list by decreasing bottom-level. We can now extract the set of Δ-Critical Tasks (Δ-CT) of size d. For each of these particular tasks we try to minimize its earliest finish time (EFT) while taking the other critical tasks into account by limiting the number of processors that be can allocated to a task. The maximal number of processors that can be allocated to a task $t_i \in \Delta$-CT on cluster c_j is then limited to $a_j = p_j/(\lfloor d/C \rfloor + b_j)$, where b_j correspond to the repartition of the $d \, mod \, C$ remaining tasks between the clusters, with regard to the relative cumulative power of the clusters. For instance, if $d = 8$ and $C = 3$, there are two tasks to dispatch among three clusters. If the heterogeneity factor of the platform is low, the first two clusters will respectively determine a_0 and a_1 considering one more task than the third cluster *i.e.,* $b_0 = b_1 = 1$, $b_2 = 0$. If the heterogeneity factor is higher and the first cluster is twice as fast as the slowest cluster of the platform, the two remaining tasks will only influence the computation of the maximal number of processors that can be allocated to a task on the first cluster, *i.e.,* $b_0 = 2$, $b_1 = b_2 = 0$. The rationale is to favor the scheduling of more concurrent tasks on the clusters having an high cumulated power.

V. Evaluation

Our goals in this section are to quantify the impact of taking several tasks with close priority levels into account in the scheduling process and therefore to demonstrate the validity of our original algorithm.

A. Experimental Methodology

We use simulation as it makes it possible to explore wide ranges of application and platform scenarios in a repeatable manner and to conduct statistically significant numbers of experiments. Our simulator is implemented using the SimGrid toolkit[1] [13], which provides the necessary fundamental abstractions and models for the discrete-event simulation of parallel applications in distributed environments. SimGrid and its SimDag interface are particularly appropriate here as it was specifically developed with the evaluation of DAG scheduling algorithms in mind.

1) Simulated Platforms: The Grid'5000[2] project aims at building a highly reconfigurable, controlable and monitorable experimental Grid platform gathering 9 sites geographically distributed in France featuring a total of 5000 CPUs. The objective of this project is to provide to the community of Grid researchers a testbed allowing experiments in all the software layers between the network protocols up to the applications. The physical platform features 9 local platforms, with at least

[1] http://simgrid.gforge.inria.fr
[2] https://www.grid5000.org

one cluster per site, each with a hundred to a thousand nodes. These nodes are either based on AMD Opteron, Intel Xeon, Intel Itanium 2 or even PowerPC architectures. The nodes inside each cluster are connected through Gigabit devices (GigaEthernet or Myrinet) and clusters are interconnected by the RENATER Education and Research Network at 10 Gigabit per second. In our simulations we consider that the interconnection network is dedicated to our experiments and without variations of availability.

The Grid'5000 platform is only representative of a certain class of computational grids, that of multi-cluster platforms located in a single administrative domain. On this kind of platforms many issues disappear, *e.g.,* firewalls or heterogeneous resource management. Scheduling mixed-parallel applications onto more complex platforms requires a more hierarchical approach (global and local for instance) to which we think our work could be adapted.

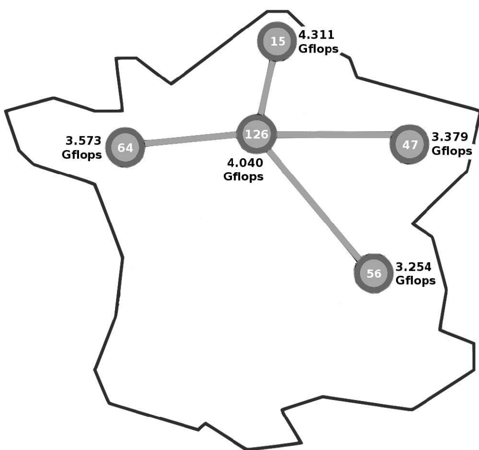

Fig. 1. Geographical repartition of our subset of the Grid'5000 platform, with numbers of nodes per site and processing speeds.

In this paper we consider a subset of Grid'5000 as a target simulated platform. We chose five clusters each being made of a different generation of AMD Opteron processors. Figure 1 shows the geographical repartition of these clusters, the number of processors in each cluster and the speed (in GFlop/s) of one processor. This platform comprises a total of 308 processors

From this platform we extract three other sub-platforms with different features onto which validate our heuristic. In the first sub-platform (*g5k_88*) each cluster has almost the same cumulated processing capacity (64-65 GFlop/s) leading to an heterogeneity factor (*i.e.,* the ratio between the fastest and the slowest clusters) of 1.37%. In the second sub-platform (*g5k_75*), all clusters have the same number of processors (15) in order to reflect the heterogeneity of single processor speeds. Here the heterogeneity factor of the platform is of 32.48%. Finally the third sub-platform (*g5k_174*) comprises two clusters with the same baseline cumulated power (Lille (64.67 GFlop/s, 15 procs.) and Nancy (64.2 GFlop/s, 19 procs.)), two clusters showing a cumulated power twice as big (Rennes (128.63 GFlop/s, 36 procs.) and Lyon (130.16 GFlop/s, 40 procs.)) and one cluster with a cumulated power four times higher than the baseline (Orsay (258.56 GFlop/s, 64 procs.)). The heterogeneity factor of *g5k_174* is of 299.81%.

2) Simulated Applications: To evaluate the relative performance of the heuristics, we first considered randomly generated application graphs. We assume that a data-parallel task operates on a data set of n double precision elements (for instance a $\sqrt{n} \times \sqrt{n}$ square matrix). Since each processor has 1 GByte of memory, we assume that n can be at most $121M$. We also assume that n is above $4M$ (if n is too small, the data-parallel task should most likely be aggregated with its predecessor or successor). The cost of data redistribution and data communication between two tasks depends on n. We model the computational complexity of a task as one of the three following forms: $a \cdot n$, $a \cdot n \log n$, $a \cdot n^{3/2}$, where a is picked randomly between 2^6 and 2^9. These complexities are inspired by common linear algebra kernels that are likely data-parallel components of a mixed-parallel application. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three. Finally, we assume that a fraction α of a task's sequential execution time is non-parallelizable [14], with α uniformly picked between 0% and 25%.

Now that we have a model for the data-parallel tasks, we must develop a model for the application's DAG. We consider applications that consist of 10, 20, or 50 data-parallel tasks. We use four popular parameters to define the shape of the DAG: width, regularity, density, and "jumps". The width determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2 and 0.8. We generate a first set of *layered* DAGs using these three parameters with the particularity that all the tasks in a given level have the same cost. Consequently all the transfers between the same two levels share the same communication cost. We have 96 different DAG types in this set. Since some elements are random, for each DAG type we generate five sample DAGs, for a total of 480 layered DAGs.

We also generate another set of irregular DAGs in which tasks in a same level can have different costs. 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). We refer the reader to our DAG generation program and its documentation for more details [15]. We have 288 different irregular DAG types and five sample of each of these types, for a total of $1,440$ irregular DAGs. Table I summarizes the different parameters used to generate our random DAGs and the associate values.

TABLE I

RANDOM DAG GENERATION PARAMETERS AND VALUES.

#computation tasks	10, 20, 50
computational complexities	$a \cdot n$, $a \cdot n \log n$, $a \cdot n^{3/2}$
non-parallelizable fraction	[0.0; 0.25]
width	0.2, 0.8
density	0.2, 0.8
regularity	0.2, 0.8
jump length	1, 2, 4
#samples	5

In addition to these randomly generated task graphs, we also considered task graphs of two real world problem: Fast Fourier Transformation [16] and Strassen's matrix multiplication algorithm [17]. For these two applications graphs the shape is fixed by the algorithms but the costs associated to computation and transfer nodes are generated following the same generation approach as for the random graphs. We generate 25 samples for each parameter combination leading to 400 FFT DAGs and 100 Strassen DAGs.

Figure 2 shows the task graph of the one-dimensional FFT algorithm [16] with four data points. This task graph can be divided in two parts corresponding respectively to the recursive calls and the butterfly operations of the algorithm. For m data points, there are $2 \times m - 1$ recursive call tasks and $m \times \log_2 m$ butterfly operation tasks. The main feature of the FFT task graph is that every path from the start node to any of the exit tasks is a critical path, *i.e.,* computation or communication tasks in a given level have the same cost. In the FFT-related experiments, we used m, the number of data points as a parameter of our simulations (2, 4, 8, and 16), to generate FFT-shaped DAGs with different number of tasks.

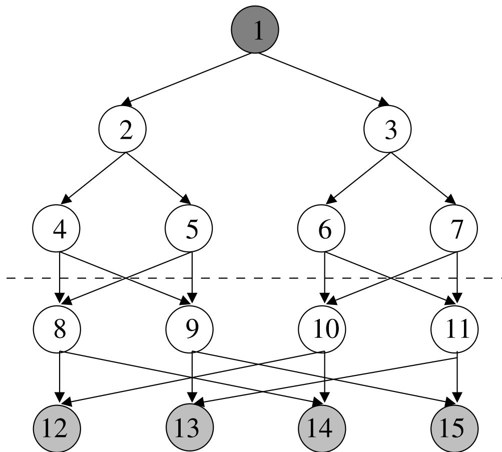

Fig. 2. Task graph of the FFT algorithm with four points.

Figure 3 shows the task graph of the Strassen's matrix multiplication algorithm [17]. As for the FFT application graph, all entry tasks are on a critical path and computation or communication tasks in a given level have the same cost.

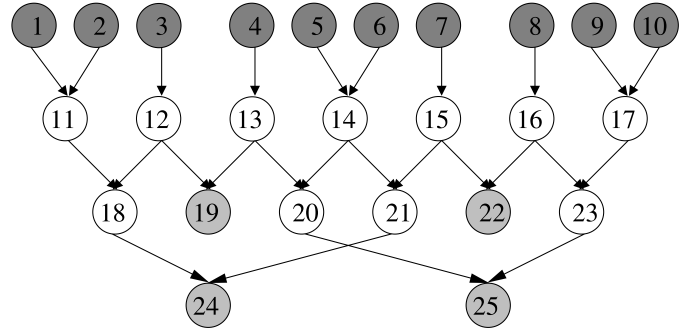

Fig. 3. Strassen's algorithm task graph .

3) Simulation Procedure: For all platform and application configurations described above, a total of $9,680$ experiments, we compare our Δ-CTS heuristic to M-HEFT and two modified versions of M-HEFT. In IMP5, a task's allocation is increased by one processor only if that task's *execution time* is improved by more than 5%. The goal is not to add processors if the task will not go significantly faster. In MAX50, no task allocation on a cluster can be larger than 50% of the total number of processors in that cluster. Algorithms are compared using the following classic metrics [11].

Schedule Length Ratio As the properties of a DAG directly impact on the schedule length (*makespan*) produced by a scheduling algorithm, there is a need to normalize this length to a lower bound, denoted as Schedule Length Ratio (SLR) and defined by

$$SLR = \frac{makespan}{CP_{min}}. \tag{1}$$

Where CP_{min} represents the summation of the execution times of tasks being in the critical path achieved on the set of processors with the maximal power. No scheduling algorithm can produce a better schedule than this summation as the denominator is a lower bound. The algorithm achieving the lowest SLR is the best with regard to this metric.

Speedup The speedup value of a graph is computed by dividing the sequential time (*i.e.,* the cumulative execution time of the tasks) achieved on the fastest processor, by the parallel execution time (*i.e.,* the schedule makespan)

$$Speedup = \frac{\sum_{t_i \in \mathcal{N}} T(t_i, 1)}{makespan} \tag{2}$$

Number of Occurrences of better Quality Schedules We count the number of times that each algorithm produces better, worse, and equal quality of schedule with regard to every other algorithm.

Degradation from Best The degradation from best measures the percent relative difference between the makespan achieved by an algorithm and the makespan achieved by the best algorithm for a given experiment.

B. Results

In this section, we compare our Δ-CTS heuristic to the original and modified M-HEFT algorithms with regard to the different metrics to evaluate the impact of considering several concurrent tasks during the allocation process.

1) Average Schedule Length Ratio: Figure 4 shows the average schedule length ratio of the scheduling algorithms depending on the target platform for all application task graphs. First we can see that the SLR increases for all algorithms along with the heterogeneity factor of the platform. As the lower bound CP_{min} used to normalize the SLR depends on the maximum cumulative power of a cluster of the target platform, its value indeed decreases as the heterogeneity grows. Then we can see that Δ-CTS has a better SLR than M-HEFT over the whole set of experiments which is the best competitor. On small platforms with a low heterogeneity factor (*g5k_88* and *g5k_75*), the schedule lengths of these two heuristics are very close, while MAX50 and IMP5 produce clearly worse schedules. As the heterogeneity increases, Δ-CTS and MAX50 becomes better than M-HEFT. This is quite obvious as leaving some room to concurrent tasks when taking a scheduling decision is efficient when platforms are sufficiently large and when more than one task at a time can benefit of high speed cluster. The added value of Δ-CTS over MAX50 is to adapt this bound to the actual shape of the DAG to share processors only when needed. Finally IMP5 shows poor performance except on the whole platform on which its smaller allocations benefit of the large and powerful cluster of Orsay, by allowing more tasks to be scheduled concurrently.

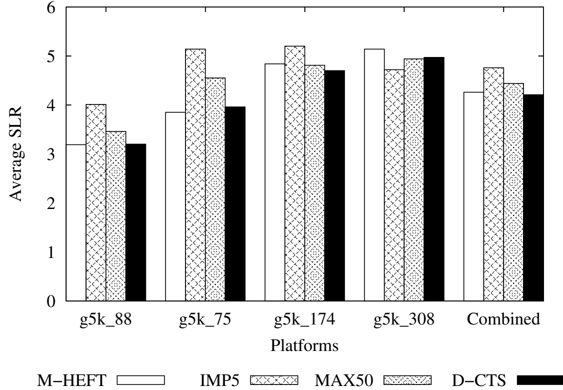

Fig. 4. Average SLR of the Scheduling Algorithms depending on the platforms for all application task graphs.

Figure 5 also shows the average schedule length ratio of the scheduling algorithms but now depending on the applications. For FFT task graphs and random layered DAGs, Δ-CTS clearly outperforms the other heuristics (8.47%-24.32% on FFT, 4.53%-10.28% on layered DAGs). For Strassen, MAX50 produces slightly better schedules (2.35%) than Δ-CTS. This can be easily explained. The width of a Strassen DAG is 10 and our platforms are made of 5 clusters, Δ-CTS thus determines the same allocation as MAX50 for the first level. But the second level is less large than the first, so Δ-CTS can allocate more processors to those tasks of the second level while MAX50 keeps the same bound. This results in more data redistribution between first and second levels in Δ-CTS than in MAX50 and thus longer schedule lengths. Finally for irregular random DAGs, no heuristic emerges as a clear winner between M-HEFT, MAX50 and Δ-CTS. The explanation is that the maximal size of the Δ-Critical Task set is less or equal to the number of clusters in the platforms for 73.3% of our irregular DAGs, and only for 56.9% of our layered DAGs. Consequently, our heuristic produces the same schedules as M-HEFT for most of the irregular DAGs.

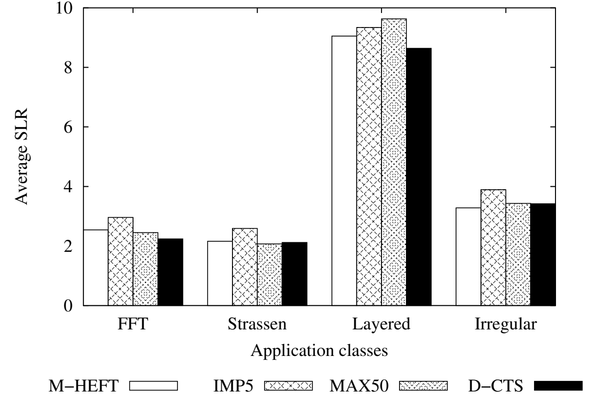

Fig. 5. Average SLR of the Scheduling Algorithms on all Platforms Depending on the Application Classes.

2) Average Speedup: Figure 6 shows the speedup achieved by the different scheduling algorithm depending on the target platforms. We can see that we have the same trends as in Figure 4. The IMP5 modified version of M-HEFT achieves the poorest speedup. Δ-CTS always has the better speedup even if on platforms with a low heterogeneity factor, the gain over M-HEFT is small. But when the heterogeneity factor increases, Δ-CTS produces better schedules and thus also has better speedups. On the whole platform, MAX50 becomes slightly better (5.42% of improvement). Over the complete set of experiments, Δ-CTS improves the speedup of 5.49% and 5.73% with regard to M-HEFT and MAX50 respectively.

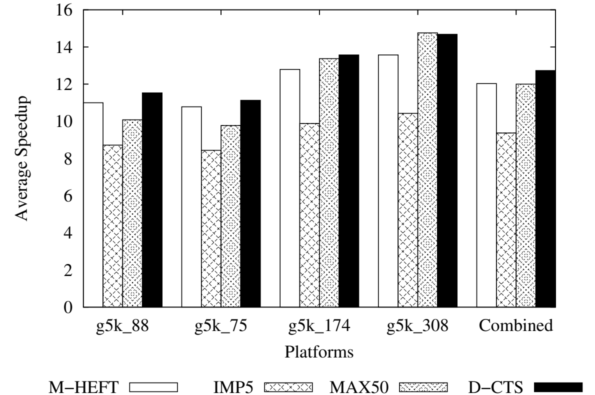

Fig. 6. Average Speedup of the Scheduling Algorithms.

TABLE II
PAIR-WISE COMPARISON OF THE SCHEDULING ALGORITHMS.

		M-HEFT	IMP5	MAX50	Δ-CTS	Combined
M-HEFT	better		8081	6492	1683	55.98%
	equal	XXX	1	2	5696	19.62%
	worse		1598	3186	2301	24.40%
IMP5	better	1598		1321	1122	13.92%
	equal	1	XXX	21	1	0.08%
	worse	8081		8338	8553	86.01%
MAX50	better	3186	8338		2672	48.88%
	equal	2	21	XXX	20	0.15%
	worse	5673	1484		6590	47.34%
Δ-CTS	better	2301	8557	6988		61.45%
	equal	5696	1	20	XXX	19.69%
	worse	1683	1122	2672		18.86%

3) Number of occurrences of Better Quality Schedules: The number of times that each scheduling algorithm produced better, equal or worse schedule length compared to every other algorithm was counted for the 9,680 experiments. Each cell in Table II indicates the comparison results of the algorithm on the left with the algorithm on the top. The *combined* column shows the percentage of scenarios in which the algorithm on the left gives a better, equal or worse performance than all other algorithms combined. The ranking of the algorithms, based on occurrences of best results, is {Δ-CTS, M-HEFT, MAX50, IMP5}. This confirms the ranking with regard to average SLR values. We can also see that Δ-CTS produces equal schedule lengths with regard to M-HEFT in 58.84% of the experiments. This percentage may seem high but can easily be explained. Indeed, when the width of a DAG is less or equal to the number of clusters in the platform (5), Δ-CTS determines the same allocations as M-HEFT. In our test plan, many DAGs have a small width. For instance, FFT DAGs with 2 and 4 points (200 task graphs), random DAGs (layered and irregular) with 10 or 20 tasks generated with the width parameter set at 0.2 (640 DAGs) and have such a width. With our four platforms, this leads to 3,360 experiments. The remaining cases come from some of the random DAGs with 50 tasks generated with the width parameter set at 0.2 and from irregular DAGs for which the size of the Δ-Critical Task set is less than five, as said before. We can thus conclude that Δ-CTS produces better or equal schedule lengths than one of the other algorithms in more than 80% of the experiments.

4) Degradation from Best: An interesting complement to this study of the number of occurrences of better quality schedules is to evaluate the degradation from best. This allows us to determine the relative quality of the schedules produced by an algorithm when these schedules are not the bests. Table III shows results obtained with two computation methods for the degradation from best. The first line presents the average over the total number of experiments (9,680) of the percent relative difference between the makespan achieved by an algorithm and the best makespan achieved for a given experiment. We can see that when Δ-CTS is not the best heuristic, the schedule lengths produced are less than 6% longer in average. Compared to M-HEFT and MAX50, the degradation from best of Δ-CTS is 32.14% and 59.37% better respectively.

TABLE III
AVERAGE DEGRADATION FROM BEST OF THE SCHEDULING ALGORITHMS.

	M-HEFT	IMP5	MAX50	Δ-CTS
Avg. on all exp.	8.43%	66.92%	15.07%	5.72%
#not-best	3868	8941	7770	3686
Avg. on #not-best	21.10%	72.45%	18.78%	15.03%

One may criticize this averaging method as if a heuristic is often the best, dividing the sum of each of its particular degradations from best – which often are 0 – by the total number of experiments biases the results. To alleviate such a critic, Table III also shows a second way to compute the average degradation from best in which the sum is divided by the number of experiments where the heuristic did not produced the best schedule length. The second line of the table shows, for each scheduling algorithm, the number of such experiments, while the third line presents the average degradation from best of each algorithm computed by that second method. Δ-CTS is still the algorithm which schedule lengths are the closest to the best. Compared to M-HEFT and MAX50, the degradation from best of Δ-CTS is 28.76% and 19.96% better respectively. Finally, it has to be noticed that Δ-CTS is the scheduling algorithm that has the minimum number of occurrences of non-best schedules.

VI. CONCLUSION

In this paper we have studied the scheduling of an application with mixed parallelism, *i.e.,* represented by a DAG of data-parallel tasks, onto a national grid that consists of multiple clusters. If list-scheduling heuristics exists for such applications they take selfish processor allocation decisions for each task of the scheduling list without considering that subsequent tasks that can be executed concurrently can be impacted by these decisions. To address this issue we proposed an original heuristic, called Δ-CTS, in which we decompose

the scheduling list into groups of tasks having almost the same priority and thus being almost as critical. For applications represented by layered DAGs, the size of such groups is closely related to the width of each level of the task graph, while in more irregular DAGs, it is more likely to be one. For a task in a Δ-CT group, the maximal number of processors that can allocated to it is then bounded to ensure that all the tasks of the group may have a chance to be executed concurrently.

We showed that our Δ-CTS heuristic improves the applications makespans with regard to the M-HEFT [5] algorithm and its improved versions [6] on a large range of application scenarios and for different real-world platform configurations. We also showed that in 80% of the experiments Δ-CTS produces better or equal schedule lengths with regard to its competitors. Finally, we concluded our evaluation by showing that when our heuristic did not produce the best schedule, the achieved makespan is still really close (less than 6% on average).

If the gain is clear for layered application DAGs, there is still room for improvement on irregular DAGs. Indeed even with considering tasks with a bottom-level priority as close as Δ, the size of concurrent tasks is often too small to see a gain with regard to the original M-HEFT algorithm. Part of our future work will consist in investigating how to find a more appropriate Δ parameter not only to produce better schedules for the random irregular DAGs of this paper but also to extend this work to the scheduling of multiple application DAGs. Some recent work [18] considered how to use HEFT to schedule multiple DAGs made of sequential tasks by aggregating the different application task graphs into one single DAGs. Extending the work presented in this article to multiple mixed-parallel applications is quite straightforward but implies to efficiently manage a scheduling list comprising tasks coming from every applications and thus with different bottom-level priorities.

Acknowledgments

The author wishes to thank the anonymous reviewers for their insightful comments.

The work presented in this paper and the development of the SimDAG interface of the SimGrid Toolkit were supported by the ARC INRIA OTaPHe.

References

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

[2] R. Lepère, D. Trystram, and G. Woeginger, "Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints," *Int. Journal on Foundations of Computer Science*, vol. 13, no. 4, pp. 613–627, 2002.

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

[4] J. Turek, J. Wolf, and P. Yu, "Approximate Algorithms for Scheduling Parallelizable Tasks," in *Symp. on Parallel Algorithms and Architectures*, 1992, pp. 323–332.

[5] H. Casanova, F. Desprez, and F. Suter, "From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling," in *10th International Euro-Par Conference*, ser. LNCS, vol. 3149. Pisa, Italy: Springer, Aug. 2004, pp. 230–237.

[6] T. N'Takpé, F. Suter, and H. Casanova, "A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms," in *6th International Symposium on Parallel and Distributed Computing (ISPDC'07)*, Hagenberg, Austria, Jul 2007.

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

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

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

[10] S. Bansal, P. Kumar, and K. Singh, "An Improved Two Step Algorithm for Task and Data Parallelism in Distributed Memory Machines," *Parallel Computing*, vol. 32, pp. 759–774, 2006.

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

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

[13] A. Legrand, L. Marchal, and H. Casanova, "Scheduling Distributed Applications: The SimGrid Simulation Framework," in *3rd IEEE Symp. on Cluster Computing and the Grid (CCGrid)*, May 2003, pp. 138–145.

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

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

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

[17] V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

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