Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications

Sascha Hunold [#1], Thomas Rauber [#2], Frédéric Suter [*3]

[#] *Department of Mathematics and Physics*
University of Bayreuth, Germany
[1] hunold@uni-bayreuth.de
[2] rauber@uni-bayreuth.de

[*] *Nancy Université / LORIA*
UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1
[3] Frederic.Suter@loria.fr

***Abstract*— Applications raising in many scientific fields exhibit both data and task parallelism that have to be exploited efficiently. A classic approach is to structure those applications by a task graph whose nodes represent parallel computations. Scheduling such *mixed-parallel* applications is challenging even on a single homogeneous platform, such as a cluster. Most of the mixed-parallel application scheduling algorithms rely on two decoupled steps: allocation and mapping. This separation can induce unnecessary or costly data redistributions that have an impact on the overall performance. This is particularly true for data intensive applications. In this paper, we propose an original approach in which the allocations determined in the first step can be adapted during the second step in order to minimize the impact of these data redistributions. Two redistribution aware mapping strategies are detailed and a study of their impact on the schedule length is proposed through a comparison with an efficient two step algorithm over a broad range of experimental scenarios.**

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. A data-parallel application typically exhibits parallelism at the level of loops. A way to expose increased parallelism, to achieve higher scalability and performance, is to write parallel applications that use both types of parallelism, using what is often called *mixed parallelism*. Mixed-parallel applications are structured as graphs of data-parallel tasks. Such structures arises naturally in many applications (see [1] for a discussion of the benefits of mixed parallelism and for application examples). A programming environment designed to express and execute mixed-parallel applications on the grid is TGrid [2]. This framework takes a task graph as input, finds ready nodes in the task graph, and maps (schedules) them onto different grid sites. Each of these nodes can be implemented in a data-parallel way, for example using MPI.

One well-known challenge for mixed-parallel applications is *scheduling*, that is making decisions for mapping computation and data transfers to platform components to optimize some performance metric. The vast majority of works that target the scheduling of mixed-parallel applications use application execution time, or *makespan*, as the performance metric. Mixed parallelism adds another level of difficulty to the already challenging scheduling problem for task-parallel applications. This is because data-parallel tasks can be moldable, *i.e.,* they can be executed on various numbers of processors, with more processors leading to faster task execution times. It raises the additional question of how many processors should be allocated to each data-parallel task.

The most popular parallel computing platforms today are commodity clusters, which are therefore primary candidates for running mixed-parallel applications. Most clusters consist of identical compute nodes (at least when they are initially put in production) and thus the question of scheduling mixed-parallel applications on *homogeneous* platforms has been studied by many researchers. Several practical scheduling algorithms based on heuristics have been proposed in the literature [3], [4], [5], [6] that proceed in two steps. In a first step, the algorithm decides how many processors should be allocated to each task, while in step two, the algorithm uses a list scheduling approach to map tasks to sets of processors. Most of these algorithms do not take data redistributions, induced by data dependences, into account during the allocation step as it is difficult to accurately estimate the redistribution times before tasks are actually mapped onto the platform. Ignoring redistributions while allocating may lead to two types of situations that impact the overall performance of the scheduling algorithm. First, subsequent tasks may have close but different allocations that may imply a complex data redistribution that could be avoided. Second, because of contention on network links, data redistributions can delay the start date of a task and thus compromise a schedule based only on task execution time reduction. This becomes particularly true for applications dominated by the data for which the communication costs cannot be neglected.

In this paper, we propose an original approach in which the allocations determined in the first step can be adapted during the second step in order to minimize the impact of the data redistributions. We first give some background and related work in Section II. Then we present several strategies

used in the proposed Redistribution Aware Two-Steps (RATS) scheduling algorithm in Section III. As our algorithm relies on the allocation procedure of the HCPA algorithm [7], we compare the schedules produced by RATS to those given by HCPA over a broad range of scenarios in Section IV. Finally we summarize the contributions and present future work in Section V.

II. Background

A. Application Model

A mixed-parallel application can be modeled as a Directed Acyclic Graph (DAG) $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{n_i \,|\, i = 1, \ldots, N\}$ 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 n_i must send to task n_j (we call n_j a *successor* of n_i and n_i a *predecessor* of n_j). An important fact is that it is assumed that the redistribution cost between subsequent tasks n_i and n_j is zero when these tasks are executed on the same set of processors. 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 $T(t, N_p(t))$ the execution time of task t if it were to be executed on an allocation comprising $N_p(t)$ processors. 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.

To model data-parallel tasks, we assume that a task operates on a dataset of m double precision elements. We arbitrarily assume that processors have at most 1GByte of memory and thus $m \leq 121M$. We also assume that m is above $4M$ (if m is too small, the data-parallel task should most likely be aggregated with its predecessor or successor). The volume of data communicated by a task to each of its children is equal to m. As we target applications for which the communications cannot be neglected, we model the computational complexity of a task (number of operations) with the following expression: $a \cdot m$, where a is picked randomly between 2^6 and 2^9, to capture the fact that such tasks often perform multiple iterations. This is representative of a certain class of applications, for instance a stencil computation on a $\sqrt{m} \times \sqrt{m}$ domain.

While the above provides a model for sequential task execution, we also need to account for parallel executions, *i.e.,* for how task execution time varies with the number of processors. We use a speedup model that is used extensively in the literature, thus allowing our results to be compared with previously published results consistently. This model is based on Amdahl's law [8] and 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 has different execution times for different numbers of processors. This performance model is monotonically decreasing, *i.e.,* the more processors are allocated to a task, the faster its execution is. We denote by ω_i the *work* of task n_i, that is the product of its execution time and the number of processors allocated to it.

Finally we assume that data are always distributed following a one dimensional block distribution, which is one of the classic distribution scheme used in High Performance Computing. For instance, if a task n_i working on an amount of data of m bytes is mapped onto p processors, each of them will own m/p bytes. Such a data distribution allows us to easily determine the communication matrix representation a data redistribution. If the task n_j, that is a successor of n_i is mapped onto q processors, it is possible to determine which amount of data each of the p sending processors have to send to each of the q receiving processors by computing the overlapping intervals between the m/p and m/q distributions.

To illustrate this technique, let consider a simple example. Task n_i is working on 10 units of data and is mapped onto $p = 4$ processors. Each of them thus own 2.5 units of data. Task n_j is mapped onto $q = 5$ processors. The corresponding communication matrix is then given by Table I.

TABLE I
COMMUNICATION MATRIX FOR A REDISTRIBUTION OF 10 UNITS OF DATA BETWEEN $p = 4$ SENDING AND $q = 5$ RECEIVING PROCESSORS.

	q_1	q_2	q_3	q_4	q_5
p_1	2	0.5			
p_2		1.5	1		
p_3			1	1.5	
p_4				0.5	2

It has to be noticed that in this example the senders and receivers constitute disjoint set of processors. When these sets have elements in common, our redistribution algorithm tries to maximize the amount of self communications.

B. Cluster Model

In this paper, we use a model representative of clusters currently deployed in experimental or production grids such as Grid'5000[1] or EGEE[2]. A cluster comprises a set of P homogeneous nodes developing a certain processing power expressed in billions of floating operations per second (GFlop/s). As we target a generic class of applications, that may not take advantage of multi-threading, we consider that each node comprises only one processing unit. We also assume that only one task can be executed on a processing unit at a time.

Each node has its own network interface connected to a private link. We assume that the communications follow the bounded multi-port model, *i.e.,* a node can send or receive data from or to several nodes but the bandwidth of its private link is then shared among the different flows.

The interconnection of the nodes within a cluster may differ depending on the size of the cluster. In small clusters (generally up to 64 nodes) all the nodes are usually connected to the same switch, while in larger clusters nodes can be

[1] http://www.grid5000.fr
[2] http://www.eu-egee.org/

located in different cabinets, each having its own switch. The cabinet switches are then connected to another switch to create a hierarchical network. Our cluster model takes these two configurations into account.

C. Related Work

As discussed earlier, most of the scheduling algorithms for mixed parallel applications on homogeneous clusters proceed in two phases [3], [4], [5], [6]. A prominent algorithm is CPA (*Critical Path and Area-based scheduling*) [4], which aims at finding the best compromise between two quantities. The first quantity is the length of the *critical path*, *i.e.,* the path in the application task graph on which the sum of the edge and node weights is maximal. We denote the length of the critical path by C_∞. The second quantity is the ratio of the *total work*, *i.e.,* $W = \sum_{i=0}^{N} \omega_i$, by the total number of processors. We denote this ratio by $\overline{W}$. The principle of the CPA algorithm is to start by allocating only one processor to each task. Therefore, initially C_∞ is larger than $\overline{W}$. Then, at each iteration, CPA adds one more processor to the task belonging to the critical path that benefits the most from this 1-processor allocation increase. The allocation process stops when C_∞ becomes smaller than $\overline{W}$. Indeed, the case $C_\infty = \overline{W}$ corresponds to an optimal trade-off because both these quantities are lower bounds of the application makespan. Depending on application and platform characteristics, CPA may lead to excessively large allocations that can prevent the concurrent execution of independent tasks. Two algorithms address this limitation. MCPA [3] limits processor allocations to ensure that all the tasks in a level of the application DAG can be executed concurrently. This algorithm is applicable only to very regular DAGs. HCPA [7], which is also applicable to heterogeneous multi-cluster platforms, employs a modified definition of $\overline{W}$ to remove the bias induced by a large number of available processors. All of these last three algorithms use a list-scheduling-based task mapping phase in which tasks are mapped to processors in order of decreasing "bottom level" (*i.e.,* distance to the graph exit task), accounting for data communication and data redistribution costs.

It has been shown that HCPA produces shorter schedules than CPA (or at least of same length) and is applicable to a larger class of applications than MCPA. Thus we chose to use the allocation procedure of HCPA as a basis for this work.

III. Redistribution Aware Mapping

As mentioned above, the totally decoupled allocation and mapping procedures of two-step scheduling algorithms may cause important data redistribution to satisfy data dependences and favor network contention. As we assume that there is no data redistribution if two subsequent tasks are mapped on the same set of processors, the main idea of this paper is to reconsider allocations determined in the first step while mapping tasks. The proposed mapping procedure acts on a list of ready tasks because when a task becomes ready, all its predecessors have already been mapped. It is thus possible to estimate accurately the respective finish time of a task using several modified allocations for mapping.

Two different strategies can be applied as shown by Figure 1. The former consists in packing a task, *i.e.,* reducing its allocation, to obtain the same number of processors as its parent task while the latter stretches the allocation, *i.e.,* allocates more processors to the task. Behind these obvious strategies lie several related issues that are addressed in the following sections.

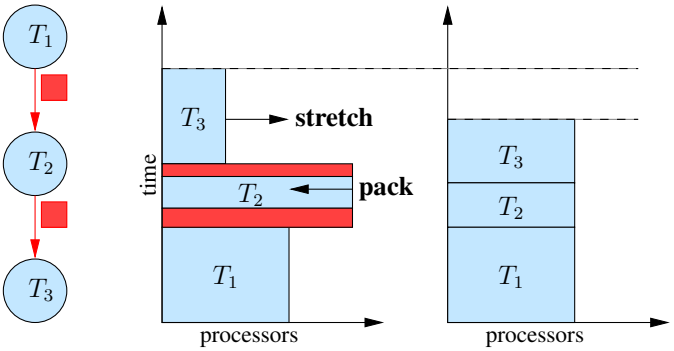

Fig. 1. Motivating example.

A. Stretching Allocations

Stretching an allocation may lead to a double gain, as it avoids a data redistribution and reduces the execution time of the task, but at the price of a higher resource usage. Furthermore this may prevent concurrent execution of ready tasks. Finally if some tasks are allocated on small sets of processors by the allocation procedure of CPA or HCPA, this means these tasks are not critical. Consequently, the number of processors that can be added to a determined allocation has to be bounded. We propose two options: the first one, named *delta*, is only concerned by avoiding a redistribution while the second, named *time-cost*, takes care of the additional work implied by the stretching of the allocation.

In the *delta* strategy, we define δ^+ as the minimal difference between the allocations of task t and that of one its predecessors: $\delta^+ = \min_i \left(N_p(pred_i(t)) - N_p(t)\right)$. We determine δ_{max}, the maximal allowed value for δ^+ on a per task basis from a parameter (called `maxdelta`) of the redistribution aware mapping procedure. This parameter takes values in $\mathbb{R}^+$ and describes the fraction of the number of processors of the original allocation that can be added. For example, if $N_p(t) = 6$ and `maxdelta = 0.5`, this means that the stretched allocation can comprise at most 9 processors $(6 + 0.5 \times 6)$ and that $\delta_{max} = 3$ for this task.

According to these definitions, when a task t is ready to schedule, the *delta* mapping procedure:

1) Check if $\delta^+ \leq \delta_{max}$,
2) Find the predecessor(s) of t corresponding to that δ^+. Keep the original allocation if there is no corresponding predecessor,
3) Map t on the same processors as those of the selected predecessor (if found).

In the *time-cost* strategy, we consider the ratio between the work corresponding to the original allocation a task and the work achieved if this task were to be executed on one of its parents' allocation:

$$\rho_i = \frac{T(t, N_p(t)) \times N_p(t)}{T(t, N_p(pred_i(t))) \times N_p(pred_i(t))}. \quad (1)$$

In addition to this ratio definition, we also have to set a threshold, ρ_{min} as a parameter of the mapping procedure to determine which allocations are candidate. This parameter takes values in the $]0 \cdots 1]$ interval. The closer ρ is to 1, the better it is, as this means a better balance between the reduction of the execution time of a task and the augmentation of the work needed for its execution. According to this definition, when a task t is ready to schedule, the *time-cost* mapping procedure:

1) Find the predecessor(s) of t producing the maximum value for ρ_i. ,
2) Check if $\rho_i \geq \rho_{min}$. Keep the original allocation if not.
3) Map t on the same processors as those of the selected predecessor (if found).

B. Packing Allocations

Packing the allocation of a task increases its execution time, as the performance model is monotonically decreasing. But this can be compensated by two consequences of such a reduction of the number of allocated processors. First, it may allow a task to start earlier as it has to wait for the availability of less processors. Then using a smaller allocation leaves more room for the execution of other potentially concurrent tasks and thus increase the exploitation of task parallelism. As for stretching we propose to apply the *delta* and *time-cost* strategies, but with slight adjustments.

In the *delta* strategy, the allocations of the predecessors candidate for packing are now larger than that of task t. We thus define $\delta^- = \max_i (N_p(pred_i(t)) - N_p(t))$. We also define δ_{min} as the minimal allowed value for δ^- on a per task basis from a parameter (called `mindelta`) of the redistribution aware mapping procedure. This parameter takes values in $\mathbb{R}^-$ and describes the fraction of the number of processors of the original allocation that can be removed. For example, if $N_p(t) = 6$ and `mindelta = -0.5`, this means that the packed allocation can comprise 3 processors at least $(6 - 0.5 \times 6)$ and that $\delta_{min} = -3$ for this task.

According to these definitions, when a task t is ready to schedule, the *delta* mapping procedure:

1) Check if $\delta^- \geq \delta_{min}$,
2) Find the predecessor(s) of t corresponding to that δ^-. Keep the original allocation if there is no corresponding predecessor,
3) Map t on the same processors as those of the selected predecessor (if found).

In the *time-cost* strategy, the mapping procedure just has to ensure that the finish time of the tasks whose allocation is packed is not worse than before packing.

C. Ready Tasks List Sorting

Another important issue is related to the order in which the ready tasks are considered for mapping. Indeed, when a task finishes its execution, more than one of its children may become ready. This raises the following question: "Which of these tasks has to be handled first?". As the different candidates have at least one predecessor in common, taking an allocation modification decision for one of them can have a negative impact on the others. For instance, stretching the allocation of a task may cause a postponing of potentially concurrent tasks by not leaving enough resources available.

As mentioned in Section II-C, the CPA and HCPA mapping procedures sort the list of ready tasks by decreasing bottom level. The rationale behind this order is that the farther a task is from the end of the application, the more critical it is and thus has to be scheduled with the highest priority. Consequently, we propose to keep this ordering of the list of ready tasks but to apply a secondary sort to order the tasks of same priority. This sort has to be stable, *i.e.,* has to keep the same order among tasks that have the same bottom level priority. We propose to apply two different sorting strategies in our redistribution aware mapping procedure. Sorting is done before mapping a ready node.

The first strategy is related to the δ^+ and δ^- parameters defined in previous sections. As δ^+ takes positive values while δ^- takes negative ones, we define $\delta(t)$ as the minimum of the δ^+ and $(-\delta^-)$ values for task t. The rationale is to prioritize tasks which require less modifications of their initial allocation. The δ sorting strategy thus applies a secondary sort to the list of the ready tasks by increasing $\delta(t)$ values.

The second strategy takes care of the time-cost tradeoff found during the allocation step. For each ready task we compute the maximal gain in terms of execution time, that can be achieved if this task were to be executed on one of its parents' processor set. We define this gain as:

$$gain(t) = \max_i \left(T(t, N_p(t)) - T(t, N_p(pred_i(t)))\right). \quad (2)$$

The *time-cost* sorting strategy applies a secondary sort to the list of the ready tasks by decreasing $gain(t)$ values.

Algorithm 1 presents the pseudo code of our redistribution aware two step scheduling algorithm.

IV. Evaluation

We use simulation for evaluating our proposed algorithm and for comparing it to previously proposed heuristics. Simulation allows us to perform a statistically significant number of experiments for a wide range of application configurations (in a reasonable amount of time). We use the SimGrid toolkit [9], [10] as the basis for our simulator. SimGrid provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms. We use SimGrid v3.3-r5344.

Algorithm 1 RATS

```
1: compute allocation /* from HCPA */
2: while not all nodes scheduled do
3:    for each ready node do do
4:       compute delta / estimate execution time
5:    end for
6:    sort ready nodes
7:    while list of ready nodes is not empty do
8:       node = pop from list of ready nodes
9:       if a parent allocation matches delta or time-cost conditions
         then
10:         map node onto parent's allocation
11:         recompute the values delta or execution time for all ready
            nodes only if they have been computed using this parent
            allocation
12:         resort ready nodes if necessary
13:      else
14:         map using HCPA
15:      end if
16:   end while
17: end while
```

A. Experimental Setup

In this paper we consider three clusters of Grid'5000 as a target simulated platform. Two of them, named *grillon* and *grelon* are located in Nancy while the third is located in Lille, named *chti*. Each cluster uses a Gigabit switched interconnect internally ($100\mu s$ latency and $1Gb$ bandwidth). The *grelon* cluster is divided into five cabinets, each comprising 24 nodes. Thus this cluster has a hierarchical network. Table II 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).

TABLE II
CLUSTER CHARACTERISTICS.

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

As our work focuses on data redistribution, it is important to present how the network is modeled within the SIMGRID toolkit. As mentioned before, SIMGRID assumes a bounded multi-port model, *i.e.,* a node can send or receive data from or to several nodes but the bandwidth of its private link is then shared among the different flows. Each network link is represented by its latency λ and its bandwidth β. To simulate gigabit networks more precisely, SIMGRID uses an empirical bandwidth $\beta' = \min(\beta, \frac{W_{max}}{RTT})$ where W_{max} is the size of the maximal TCP window and RTT is the round trip time between the computers. In case of multi-hop connection, the RTT is twice the sum of the respective latency of the different links. Finally SIMGRID models the sharing of network resources among the different communication flows implementing Max-Min fairness. A quantitative comparison between the communication model of SIMGRID and packet-level simulators can be found in [11].

To evaluate the benefits of reconsidering allocations during the mapping step, we use four types of applications and rely on parallel task model presented in Section II-A. We first consider two kinds of randomly generated application DAGs: layered and irregular. In layered DAGs, 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 while in irregular DAGs tasks that belong to a same level may have different costs. This allows us to capture the heterogeneous and unpredictable aspects of scientific workflows.

For both kinds, we generate applications that consist of 25, 50 or 100 data-parallel tasks. We use three popular parameters to define the shape of each DAG: width, regularity and density. The width determines the maximum parallelism in a 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 a 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 for density and regularity and 0.2, 0.5 and 0.8 for width. Furthermore, for irregular DAGs only, 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). Since some elements are random, for each DAG type we generate 3 sample DAGs. We refer the reader to our DAG generation program and its documentation for more details [12]. Table III summarizes the different parameters used to generate our random DAGs and the associate values.

TABLE III
RANDOM DAG GENERATION PARAMETERS AND VALUES.

	Layered	**Irregular**
#computation tasks	25, 50, 100	
non-parallelizable fraction	[0.0; 0.25]	
width	0.2, 0.5, 0.8	
density	0.2, 0.8	
regularity	0.2, 0.8	
jump length	-	1, 2, 4
#samples	3	
Total	108	324

In addition to these randomly generated task graphs, we also considered task graphs of two High Performance Computing kernels: Fast Fourier Transformation and Strassen's matrix multiplication algorithm. 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 100 FFT DAGs and 25 Strassen DAGs.

The FFT task graph can be divided in two parts corre-

sponding respectively to the recursive calls and the butterfly operations of the algorithm. For k data points, there are $2 \times k - 1$ recursive call tasks and $m \times \log_2 k$ 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 k, 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 (5, 15, 39 and 95).

As for the FFT application graph, all the entry tasks of the Strassen's matrix multiplication algorithm are on a critical path and computation or communication tasks in a given level have the same cost. A Strassen DAG comprises 25 tasks.

B. Impact of Redistribution Aware Mapping

We measure the impact of the redistribution aware mapping as follows. For the 557 application configurations, we compute a schedule using RATS (Redistribution Aware Two Step) and one using HCPA [7]. For each schedule we compute its makespan (lower values mean better performance) and its total work (lower values mean lower resource consumption).

Two versions of RATS are compared. The first version relies on the *delta* strategy, *i.e.,* aims at avoiding one data redistribution per task by changing the original allocation by at most a factor δ and sorting the ready tasks by increasing values of δ as explained in the previous section. The second version adopts a *time-cost* strategy that stretches an allocation only if the ratio between execution time and work done is improved or pack the allocation only if the task can finish earlier and sort the ready tasks according to the gain that can be achieved. For this first comparison, we use a naive value (0.5) for each parameter. For `mindelta` (resp. `maxdelta`), this value means that an allocation can be at most decreased (resp. increased) by 50%. For `minrho`, the efficiency loss can be at most of 50%.

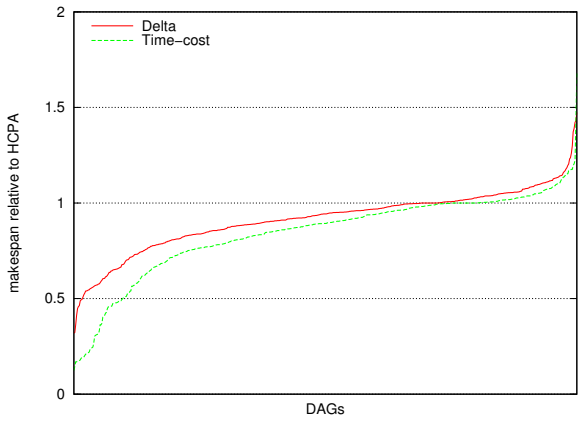

Fig. 2. Relative makespan of RATS using the *delta* (`mindelta` = `maxdelta` = 0.5) and the *time-cost* (packing allowed and `minrho` = 0.5) strategies compared to HCPA on the *grillon* cluster.

Figure 2 shows the makespan achieved for each application configuration by the two versions of RATS relative to that achieved by HCPA on the *grillon* cluster. The data points are sorted by increasing value of this relative makespan. Note that the data sets are sorted independently. We see that across our application configurations the makespan achieved by the *delta* strategy is on average 9% shorter than HCPA and leads to shorter schedules in 72% of the scenarios. We can also see that the *time-cost* strategy leads to better results as on average this strategy leads to makespans 16% shorter than HCPA and to shorter schedules in 80% of the scenarios. We did not find any particular trends in this data with respect to application configuration characteristics. Furthermore, similar results were obtained on the *chti* and *grelon* clusters.

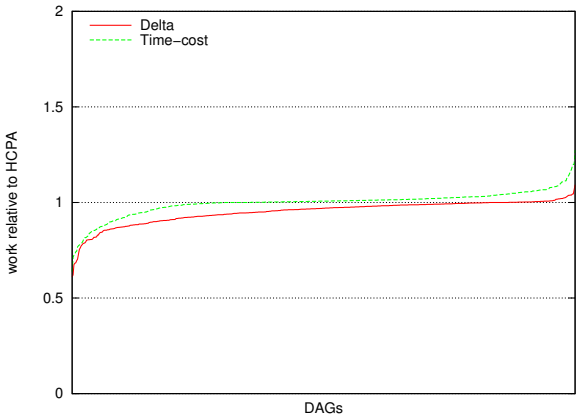

Fig. 3. Relative work of RATS using the *delta* (`mindelta` = `maxdelta` = 0.5) and the *time-cost* (packing allowed and `minrho` = 0.5) strategies compared to HCPA on the *grillon* cluster.

Similarly, Figure 3 shows the total work of the schedules produced on the *grillon* cluster by RATS relative to that of schedules produced by HCPA for all application configurations. We see that overall both RATS versions do not consume much more resources than HCPA. We can also see that the *delta* strategy consumes less resources than the *time-cost* which is coherent with the better makespans achieved by the *time-cost* strategy. Again, similar results were obtained on the *chti* and *grelon* clusters.

C. Tuning δ and ρ Parameters

In this section we describe how the behavior of the two versions of RATS can be tuned to trade off its resource usage for average performance (*i.e.,* obtain a lower average makespan over our range of mixed-parallel applications).

In the *delta* strategy, two parameters (`mindelta` and `maxdelta`) have an influence on the allocation modifications. In the previous section we fixed them to 0.5. We now try to determine which pair of values allows to achieve the smallest makespan relative to HCPA, for each cluster but also for each type of applications. Figure 4 presents the methodology we used and the results obtained for FFT DAGs on the *grillon* cluster. For both parameters we tested 4 values: 0, 0.25, 0.5 and 0.75. Another value (1) was tested for `maxdelta` only, as allowing to remove all the processors of an allocation when packing does not make sense. Then we compute the

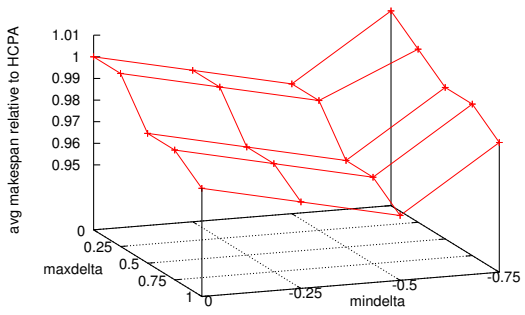

Fig. 4. Relative makespan of RATS using the *delta* strategy for FFT DAGs compared to HCPA on the *grillon* cluster as (`mindelta`, `maxdelta`) vary.

average makespan relative to that achieved by HCPA for each combination of `mindelta` and `maxdelta`.

We can see on that particular example that changing for larger allocations (allowed by a larger `maxdelta`) leads to a better average relative makespan. It can be easily explained by the possibility to use more resources to execute a given task. On the other hand, decreasing the value of `mindelta` can also improve a schedule, mainly because more tasks can be executed in parallel with smaller allocations, but only to a certain extent. Furthermore, the impact of the value taken by `mindelta` is more sensitive to the application characteristics as shown in Table IV.

In the *time-cost* strategy, there are also two variable parameters but one is a boolean enabling or disabling the possibility to pack allocations. We noticed that setting this boolean to true (*i.e.,* enable allocation packing) always produces shorter schedules. It has to be recalled that in the *time-cost* strategy an allocation is packed if and only if the finish time of the corresponding task is reduced. Allowing the mapping procedure to pack allocations can thus sparsely have a negative impact on the schedule length.

The other parameter is `minrho` and acts on allocation stretching. As for `mindelta` and `maxdelta`, we have tested several values (0.2, 0.4, 0.5, 0.6, 0.8 and 1) and determined which value leads to the best average relative makespan for each cluster-application type pair. Small values of `minrho` allows for more flexibility as allocations can be stretched even if the time-cost ratio between initial and stretched allocations is not very good.

Figure 5 details a particular example (irregular random DAGs on the *grillon* cluster) to show that allowing allocations to be packed gives better performance and a threshold can be found (here 0.5), and beyond that, there is no need to increase the flexibility.

Table IV summarizes the different values of `mindelta`, `maxdelta` and `minrho` that have been determined for each application type and cluster. These values will be used in the next section to perform another comparison between the two versions of RATS and HCPA.

TABLE IV
VALUES OF THE RATS PARAMETERS (`MINDELTA`,`MAXDELTA`,`MINRHO`) DEPENDING ON APPLICATION TYPE AND CLUSTER.

	FFT	Strassen	Layered	Random
chti	(-.5, 1, .2)	(-.25, .5, .5)	(-.5, 1, .2)	(-.75, 1, .5)
grillon	(-.5, 1, .2)	(0, 1, .4)	(-.25, 1, .2)	(-.75, 1, .5)
grelon	(-.25, .75, .4)	(-.25, 1, .5)	(-.5, 1, .2)	(-.75, 1, .4)

D. Comparing Tuned RATS to HCPA

In this section we complete the study of the impact of a redistribution aware mapping on schedule length started in Section IV-B. Instead of using naive values, we now rely on the tuned values given in Table IV for the following experiments.

We first present the same graphs as in Section IV-B in Figures 6 and 7 that show the makespan and work achieved for each application configuration by the two versions of RATS relative to that achieved by HCPA on the *grillon* cluster. Both data sets have been sorted independently, and thus, cases exist where the *delta* strategy outperforms the *time-cost* approach.

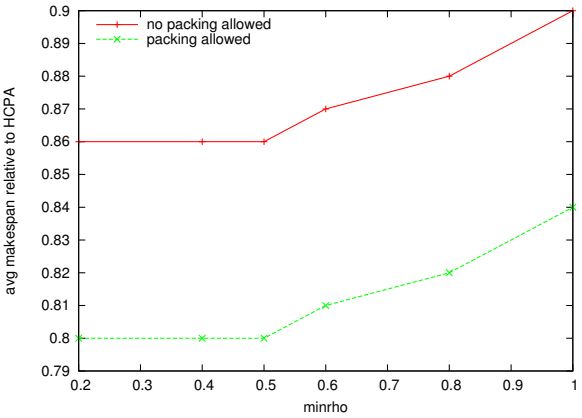

Fig. 5. Relative makespan of RATS using the *time-cost* strategy for irregular random DAGs compared to HCPA on the *grillon* cluster when `minrho` varies.

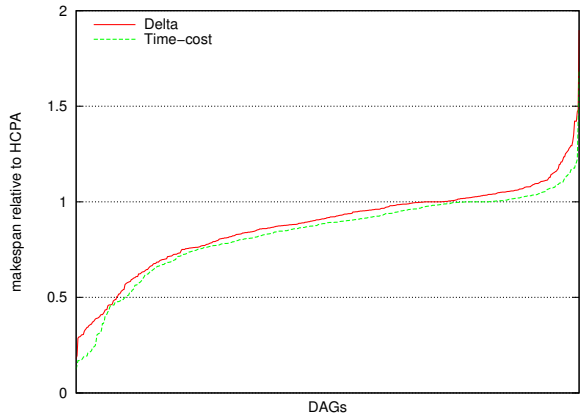

Fig. 6. Relative makespan of RATS using the *delta* and the *time-cost* strategies with tuned values compared to HCPA on the *grillon* cluster.

TABLE V

PAIR-WISE COMPARISON OF THE SCHEDULING ALGORITHMS. EACH CELL CONTAINS THE VALUES FOR *chti* / *grillon* / *grelon*.

		HCPA	*delta*	*time-cost*	combined (in %)
HCPA	better	XXX	154 / 133 / 161	103 / 88 / 82	23.1 / 19.8 / 21.8
	equal		17 / 45 / 27	21 / 50 / 22	3.4 / 8.5 / 4.4
	worse		386 / 379 / 369	433 / 419 / 453	73.5 / 71.6 / 73.8
delta	better	386 / 379 / 369	XXX	188 / 199 / 128	51.5 / 51.9 / 44.6
	equal	17 / 45 / 27		49 / 90 / 40	5.9 / 12.1 / 6.0
	worse	154 / 133 / 161		320 / 268 / 389	42.5 / 36.0 / 49.4
time-cost	better	433 / 419 / 453	320 / 268 / 389	XXX	67.6 / 61.7 / 75.6
	equal	21 / 50 / 22	49 / 90 / 40		6.3 / 12.6 / 5.6
	worse	103 / 88 / 82	188 / 199 / 128		26.1 / 25.8 / 18.9

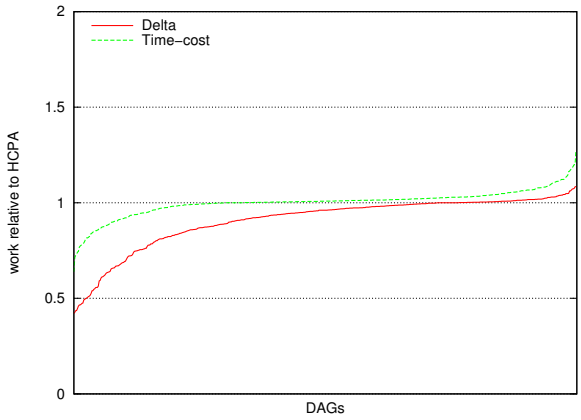

Fig. 7. Relative work of RATS using the *delta* and the *time-cost* strategies with tuned values compared to HCPA on the *grillon* cluster.

We can see that there is only a slight improvement for the *time-cost* strategy, which is normal as packing was already allowed in Figure 2 and 0.5 was a appropriate value in most scenarios. The impact of tuning is more significant for the *delta* approach. In the case of the *grillon* cluster, schedules are now 13% shorter (9% without tuning) and RATS leads to shorter schedules in more cases. Similar improvements are made on the other clusters as schedules were 8% shorter and now are 11% shorter.

It has to be noticed that this improvement in terms of makespan is not made to detriment of the resource usage. Even if allocations can be more stretched (as `maxdelta` is larger), the *delta* strategy still consumes less resources than HCPA in the vast majority of scenarios.

In addition to this comparison, the number of times that each scheduling algorithm produced better, equal or worse schedule length compared to every other algorithm was counted for the 557 experiments. Each cell in Table V indicates the comparison results of the algorithm on the left with the algorithm on the top respectively on *chti* / *grillon* / *grelon*. 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 {*time-cost*, *delta*, HCPA} which confirms the data of Figure 6.

We can also see that the *time-cost* strategy achieves better results as the size of the cluster grows while the *delta* strategy produces better schedules on small and medium sized clusters. One possible explanation is that the estimations of the redistribution time made in the *time-cost* version do not take network contention into account. For a given application, contentions are more likely to occur on a small cluster than on a lager one. Consequently, the decisions taken by RATS becomes more accurate when the size of the cluster grows. Another possible explanation is that the bigger the cluster is the more possible targets for mapping an allocation exist. Since the *delta* approach does not rely on performance estimations when mapping tasks, the introduced error while mapping allocations will have a bigger impact if many processors are available.

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 VI shows results obtained with two computation methods for the degradation from best. The first line presents the average over the total number of experiments (557) 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 the *time-cost* version of RATS is not the best heuristic, the schedule lengths produced are less than 6% longer in average. This percentage even decreases as the size of the cluster grows. Conversely, the schedules produced by the *delta* version of RATS get farther from the best ones as the cluster size grows.

TABLE VI

AVERAGE DEGRADATION FROM BEST.

		HCPA	*delta*	*time-cost*
chti	avg over all exp.	26.19 %	6.60 %	5.76 %
	# not best	453	299	239
	avg over # not best	61.03 %	15.39 %	13.42 %
grillon	avg over all exp.	45.97 %	13.87 %	5.16 %
	# not best	465	361	229
	avg over # not best	111.81 %	33.74 %	12.54 %
grelon	avg over all exp.	51.71 %	19.31 %	2.74 %
	# not best	478	412	165
	avg over # not best	174.57 %	65.18 %	9.24 %

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 VI 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. The degradation remains excellent for the *time-cost* version (less than 15%) while that of HCPA reaches very high values, *i.e.,* produces schedules that are more than twice as long as the best one.

V. Conclusion

Two-step algorithms for scheduling mixed-parallel applications on clusters structured as a task graph whose nodes are data-parallel computations have been developed [3], [4], [5], [6], [7]. However these algorithms separate the determination of the number of processors to allocate to each task from the process that decides on which processors of the cluster each task is to be executed. In this paper, we set out to develop RATS, a scheduling algorithm that addresses the issues related to data redistributions inherent to two step algorithms by reconsidering the allocations determined in the first step during the second step. Two tunable redistribution aware mapping procedure were proposed and we assessed their impact on the length of the produced schedules over a broad range of application configurations. The former strategy tries to avoid one data redistribution per task by using the processor set of a predecessor if the modification is less than a certain δ. The latter strategy modifies allocations only if the *time-cost* ratio is preserved. Experiments has shown that the proposed redistribution aware mapping procedures reduce the completion time of the applications in a vast majority of scenarios. Moreover the *time-cost* strategy leads to better results and stays very close to the best solution when it is not already the best one. Finally we shown that the tunable parameters allows the scheduling algorithm to adapt itself to platform and application characteristics to produce shorter schedules.

As a future work we aim at extending this work to multi-cluster platforms in which heterogeneity and high latency network connections have to be taken into account. We also plan to further analyze the relationships between applications and platform characteristics and our tunable parameters to allow the automatic tuning of our scheduling algorithm.

References

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

[2] S. Hunold, T. Rauber, and G. Rünger, "TGrid – Grid Runtime Support for Hierarchically Structured Task-parallel Programs," in *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Heteropar'06)*. Barcelona, Spain: IEEE Computer Society Press, Sept. 2006.

[3] 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.

[4] 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.

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

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

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

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

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

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

[11] K. Fujiwara and H. Casanova, "Speed and Accuracy of Network Simulation in the SimGrid Framework," in *First International Workshop on Network Simulation Tools (NSTools)*, Nantes, France, Oct. 2007.

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