Ocean-Atmosphere Application Scheduling within DIET

Yves Caniou, Eddy Caron, Ghislain Charrier, Frédéric Desprez, Eric Maisonnave, Vincent Pichon

Ocean-Atmosphere Application Scheduling within DIET

Yves Caniou — Eddy Caron — Ghislain Charrier — Frédéric Desprez — Éric Maisonnave — Vincent Pichon

N° 6836

February 2009

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6836--FR+ENG

Ocean-Atmosphere Application Scheduling within DIET

Yves Caniou , Eddy Caron , Ghislain Charrier , Frédéric Desprez , Éric Maisonnave , Vincent Pichon

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 6836 — February 2009 — 35 pages

Abstract: In this report, we tackle the problem of scheduling an Ocean-Atmosphere application in an heterogeneous environment. The application is used for long term climate forecast. In this context, we analyzed the execution of an experiment. An experiment is composed of several identical simulations composed of parallel tasks. On homogeneous platforms, we propose a heuristic and its optimizations, all based on the same idea: we divide the processors into disjoint sets, each group executing parallel tasks. On heterogeneous platforms the algorithm presented is applied on subsets of simulations. The computation of the subsets is done greedily and aims at minimizing the execution time by sending each subset on a cluster. We performed experiments on the french research grid *Grid'5000* which exhibited some technical difficulties. We also present some modifications done to the heuristics to minimize the impact of these technical difficulties. Our simulations are then validated by experimentations.

Key-words: Grid computing, Ocean-Atmosphere application, Scheduling, Mixed parallelism

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Ordonnancement de l'application Ocean-Atmosphere dans DIET

Résumé : Dans ce rapport, nous proposons une solution au problème d'ordonnancement dans un environnement hétérogène d'une application de modélisation Océan-Atmosphère. Cette application est utilisée pour les prévisions d'évolution du climat. Dans ce contexte, nous avons commencé par réaliser une analyse du déroulement d'une expérience de l'application. Une expérience se décompose en un ensemble de simulations indépendantes et identiques composées de tâches parallèles. Sur des plateformes homogènes, nous proposons une heuristique et ses optimisations, toutes basées sur le même principe : diviser les processeurs en ensembles disjoints, chaque groupe exécutant une tâche parallèle. Dans le cadre d'une plateforme hétérogène, l'algorithme précédent est appliqué sur des sous-ensembles des simulations. Le calcul des sous-ensembles est réalisé de manière gloutonne et cherche à minimiser le temps d'exécution en affectant un sous-ensemble à un cluster. Nous avons effectué des expériences sur la grille de recherche française *Grid'5000*, ce qui nous permet de souligner les difficultés de mise en oeuvre. Nous présentons aussi des modifications apportées aux heuristiques afin de minimiser l'impact des difficultées techniques sur le temps d'exécution d'une expérience. Nos simulations sont ensuite validées par expérimentations.

Mots-clés : Calcul sur grille, Application Ocean-Atmosphere, Ordonnancement, Parallélisme mixte

1 Introduction

World's climate is currently changing due to the increase of greenhouse gases in the atmosphere. Climate fluctuations are forecasted for the years to come. For a proper study of the incoming changes, numerical simulations are needed, using General Circulation Models (GCM) of a climate system (atmosphere, ocean, continental surfaces) on forced or coupled mode (*i.e.*, allowing information exchanges between each component during simulation).

Imperfection of the models and global insufficiency of observations make it difficult to tune model parametrization with precision. Uncertainty on climate response to greenhouse gases can be investigated by performing an ensemble prediction. Within this probabilistic approach, a set of simulations (or scenarios) with varying parameters must be launched. Each scenario models the evolution of the present climate (1950-2007) followed by the 21^{st} century.

As shown in [5] with the Met Office Unified Model GCM, the parameter with the most notable effect on climate sensitivity is the entrainment coefficient in clouds. In this way, with our GCM (ARPEGE Météo-France atmospheric model coupled with CNRS NEMO oceanic model), each simulation has a distinct physical parametrization in clouds dynamics, with a different convective nebulosity coefficient. This coefficient plays a similar role in ARPEGE cloud parametrization than entrainment coefficient for the Met Office Unified Model.

Comparing these independent scenarios, we expect to have a better understanding of the relations between the variation in this parametrization with the variation in climate sensitivity to greenhouse gases.

Our goal regarding the climate forecasting application is to continue the work started in [2]. We thoroughly analyze the application in order to model its needs in terms of execution model, data access pattern, and computing needs. Once a proper model of the application is derived, appropriate scheduling heuristics are proposed, tested, compared, and then implemented within the DIET middleware. Once the implementation done, we compare the simulations with real experiments realized on the grid.

The remainder of this paper is as follows. After a presentation of the application and the middleware which will be used in Section 2, we present a quick overview of some related work in Section 3. Then, in Section 4, we present the heuristic methods we developed for Ocean-Atmosphere. Section 5 shows the results of simulations done to observe the behavior of our heuristics. Finally, in Section 6, we present real experiments performed on the grid and the problems that arise with them before concluding in Section 7.

2 Software architecture

Ocean-Atmosphere provides weather simulations for the years to come while DIET is a GridRPC middleware. We are using an application and a middleware, so we present them separately. The integration of Ocean-Atmosphere within DIET will be presented in Section 4.3.

2.1 Application: Ocean-Atmosphere

The proposed climate application consists of executing independant simulations of present climate followed by the 21st century, for a total of 150 years (scenario). A scenario combines 1800 simulations of one month each (150×12), launched one after the other. The results from the nth monthly simulation are the starting point of the (n+1)th monthly simulation. For the whole experiment, several scenarios must be performed. The number of months to be simulated and the number of scenarios are chosen by the user.

A monthly simulation can be divided into a pre-processing phase, a main-processing parallel task, and a post-processing phase of analysis. Figure 1 shows the different tasks during the execution of a month simulation (nodes) and the data dependencies between two consecutive months (edges). The number after the name of each task represents a possible duration of the tasks in seconds. These times have been benchmarked on a given cluster and can obviously change if the resources change (power, number of nodes, ...).

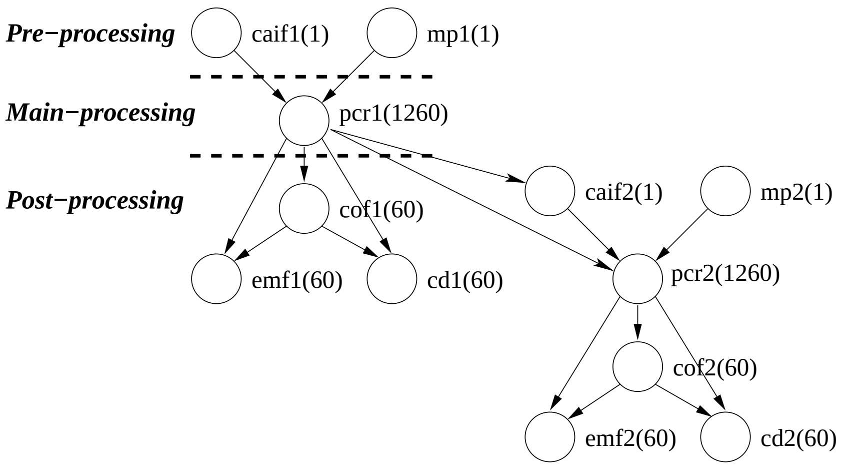

Figure 1: Chain of two consecutive monthly simulations.

During the **pre-processing phase**, input files are updated and gathered in a single working directory by **concatenate_atmospheric_input_files** (caif) and the model parametrization is modified by **modify_parameters** (mp). The whole-processing phase only takes few seconds.

The main computing task **process_coupled_run** (pcr) performs a one month long integration of the climate model. This model is composed by an atmosphere (ARPEGE [4], model belonging to Meteo-France and derived from their weather forecast version), an ocean and its sea-ice (OPA/NEMO [7], developed by CNRS at LOCEAN laboratory and shared by several european climate models), and a river runoff model (TRIP [8]). The OASIS coupler [16] ensures simultaneous run of each element and synchronizes information exchanges.

ARPEGE code is fully parallel (using the MPI communication library), while OPA, TRIP, and the OASIS coupler are sequential applications (in the chosen configuration of our climate model). The execution time of **process_coupled_run** depends on the number of processors allocated to the atmospheric model. We can note that with more than 8 processors allocated to ARPEGE the acceleration stops. OPA, TRIP and OASIS each need a processor, so pcr needs from 4 to 11 processors to be able to work. We found this limit in the number of processors for ARPEGE while testing the application. When more processors are added, the time does not decreases, but if too many processors are allocated to the task, the time increases.

The **post-processing phase** consists of 3 tasks. First, a conversion phase **convert_output_format** (cof) where each diagnostic file coming from the different elements of the climate model is standardized in a self-describing format. Then, an analysis phase **extract_minimum_information** (emi) where global or regional means on key regions are processed. Finally, a compression phase **compress_diags** (cd) where the volume of model diagnostic files is drastically reduced to facilitate storage and transfers.

Data exchanges between two consecutive monthly simulations belonging to the same scenario reach 1 GB. The post-processing phase creates an archive with results of the month execution that can be interpreted to predict the climate. The size of this archive is almost 120 MB. Simulations are independent, so no other data is used.

2.2 Middleware: DIET

DIET[1] [3] is a GridRPC middleware relying on the client/agent/server paradigm. A client is an application which needs a computing service. The agent, which can be extended as a tree hierarchy of agents, has the knowledge of several servers. There are two kinds of

[1] `http://graal.ens-lyon.fr/DIET`

agents: the MA (Master Agent) and the LA (Local Agent). The MA is at the top of the hierarchy and LAs are connected to other agents forming the tree hierarchy. The distributed scheduler embedded in each agent chooses the best computing resource for the execution of a given request. The SeD (Server Daemon) is running on the computing resource. A client can find a SeD using the DIET architecture. The SeD gives performance estimations to the agent responsible of it and launches a service when contacted by the client. Performance estimations can be described by the SeD programmer and returned through the hierarchy up to the client.

The path followed by a request is shown in Figure 2: when an agent is contacted by a client who wants to solve a problem (1), the request travels down the hierarchy of agents to servers (2). They answer back performance information (3) which will be used up in the hierarchy to determine which one suits the best (4). The identity of the server is given to the client (5) who will contact the server and send its data (6). Once the computation is finished, results are transferred back to the client (if needed).

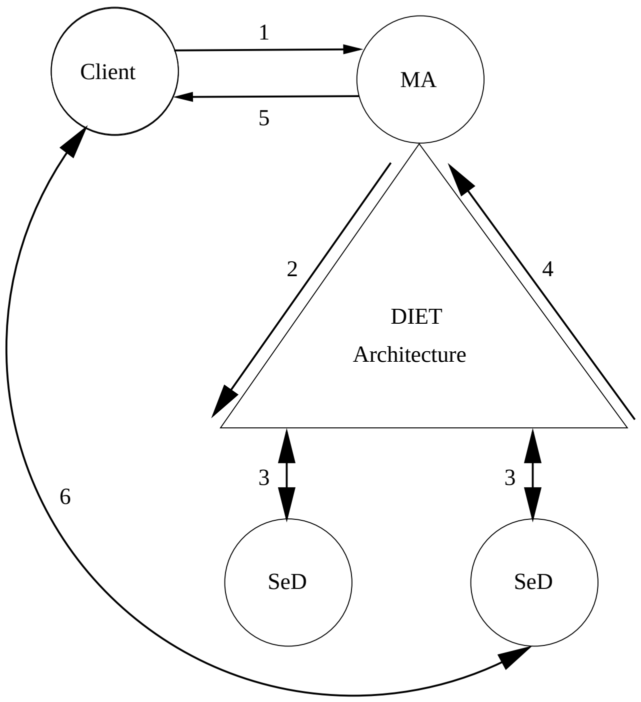

Figure 2: A DIET architecture.

3 Related Work

The execution of our application is represented by the execution of multiple DAGs containing sequential tasks and data parallel tasks. In this section, we present some existing algorithms related to our work.

To schedule multiple DAGs, authors in [17] present different methods. First, it is possible to schedule DAGs one after another on the resources. Another possibility is to concurrently schedule the DAGs. It is also possible to link all the entry tasks of the DAGs to an unique entry node and do the same with the exit nodes. Then, the new resulting DAG is scheduled.

When scheduling application using mixed parallelism (task and data parallelism), [11] proposes a two steps approach. First, the number of processors on which a data-parallel task should be executed is computed, and then, a list scheduling heuristic is used to map the tasks onto processors. In [12], an approach of scheduling task graphs is proposed. For series compositions, the tasks are allocated the whole set of processors, while for parallel compositions, the processors are partitioned into disjoint sets on which the tasks are scheduled. In [9], a one step algorithm is proposed (Critical Path Reduction - CPR) for scheduling DAGs with data-parallel tasks onto homogeneous platforms. This algorithm allocates more and more resources to the tasks on the critical path and stops once the makespan is not improved anymore. In [10], a two steps algorithm is proposed (Critical Path and Area based Scheduling - CPA). First the number of processors allocated to each data-parallel tasks is determined. In the second step, the tasks are scheduled on resources using a list scheduling heuristic.

On pipelined data parallel tasks, authors in [14] propose a dynamic programming solution for the problem of minimizing the latency with a throughput constraint and present a near optimal solution to the problem of maximizing the throughout with a latency constraint on homogeneous platforms. Several aspects must be kept in mind when mapping the tasks of a pipeline on the resources. Subchains of consecutive tasks in the pipeline can be clustered into modules (which could thus reduce communications and improve latency) and the resources can be splitted among the resulting modules. Resources available to a module can be divided into several groups, on which processes will alternate data sets, improving the throughput but reducing the latency.

The algorithm we will present here is close to the ones presented here. We will first compute the best grouping of processors, and then map tasks on them. The simulation the less advanced with a waiting task will be scheduled. Algorithms such as CPA and CPR are not applicable here because they would create as many groups as simulations. We will show that the speedup of our application is super-linear, so, having as many groups as scenarios will be a bad solution. Therefore, the algorithms presented briefly in this section will lead to bad results.

4 Scheduling for Ocean-Atmosphere

Given the really short duration of the pre-processing phase compared to the execution time of the main-processing task, we made the decision to merge them all in a single task. The same decision was taken for the 3 post-processing tasks. So, in regard of the model, there are now 2 tasks: the main-processing task and the post-processing task. Figure 3 presents the new dependencies between tasks after merging them together.

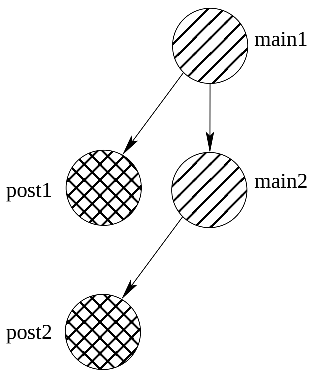

Figure 3: Chain of 2 consecutive monthly simulations after merging tasks.

In most grids, different clusters are available with different processing powers. Table 1 presents the hardware used by 7 clusters of Grid'5000[2] [1]. Grid'5000 is a grid composed of several clusters. Each cluster is composed of homogeneous resources but differs from one another. The clusters either have 2 or 4 processors per node (mono-cores bi-processors or bi-cores bi-processors).

Figure 4 shows the time needed to compute a main-task on the clusters presented in Table 1 depending on the number of resources. The times have been obtained by performing benchmarks on clusters of Grid'5000. The execution time of any task is assumed to include the time to access the data, the time to redistribute it to processors (in case the task is a multiprocessor one), the computing time, and the time needed to store the resulting data. We can see that the speedup of a main-task is superlinear: when doubling the number of resources, the time needed to execute a main task is divided by more than two. *e.g.*, with

[2]https://www.grid5000.fr

Cluster	Processor Type	Nodes	Cores	Mem.
Capricorne	AMD Opt. 246 2.0 GHz	56	112	2 GB
Sagitaire	AMD Opt. 250 2.4 GHz	70	140	2 GB
Chicon	AMD Opt. 285 2.6 GHz	26	104	4 GB
Chti	AMD Opt. 252 2.6 GHz	20	40	4 GB
Grillon	AMD Opt. 246 2.0 GHz	47	94	2 GB
Grelon	Intel Xeon 1.6 GHz	120	480	2 GB
Azur	AMD Opt. 246 2.0 GHz	72	144	2 GB

Table 1: Clusters hardware description.

5 processors on grillon, 4205 seconds are needed and with 10 processors, only 1502. The corresponding speedup is almost 2.80. This superlinear speedup comes from the fact that a main-task needs at least 4 processors, since three are used by TRIP, OPA and OASIS, and the remaining resources are used by ARPEGE. So, with 4 processors, only one is used by ARPEGE, but with 8 resources, 5 are used. This explain the superlinear speedup of a main-task. Figure 4 does not plot the execution time with more than 11 processors, but the times would be the same as for 11 resources (or greater in some cases).

To be able to work at the cluster level (homogeneous) and at the grid level (heterogeneous), we are going to present heuristics designed, first, to operate on homogeneous platforms (Section 4.1), and then, add another algorithm to make a distribution of the scenarios onto heterogeneous platforms (Section 4.3).

4.1 Scheduling on an Homogeneous Platform

The goal we want to achieve here is to minimize the overall makespan and also to keep some fairness, meaning we want all scenarios to progress in their execution at almost the same speed. We consider a homogeneous platform composed of R resources and that data on a cluster are available to all of its nodes.

The idea of this scheduling algorithm is to divide the resources of the platform into disjoint sets on which multiprocessor tasks will be executed such that the overall makespan will be minimal. We assume that all multiprocessor tasks will be executed on the same number of processors. Because the speedup of main-tasks is really good, we will not use classical algorithms that execute the maximum number of task concurrently and use more resources if the number of tasks is too small.

In [9, 10], the authors propose to give more and more processors to the critical path of the application. Since all our DAGs are the same, and tasks inside the DAGs are also the same, we will not choose how many resources to give to a DAG, but choose which task of a DAG will go on a given group of resources. To obtain fairness, when a group of

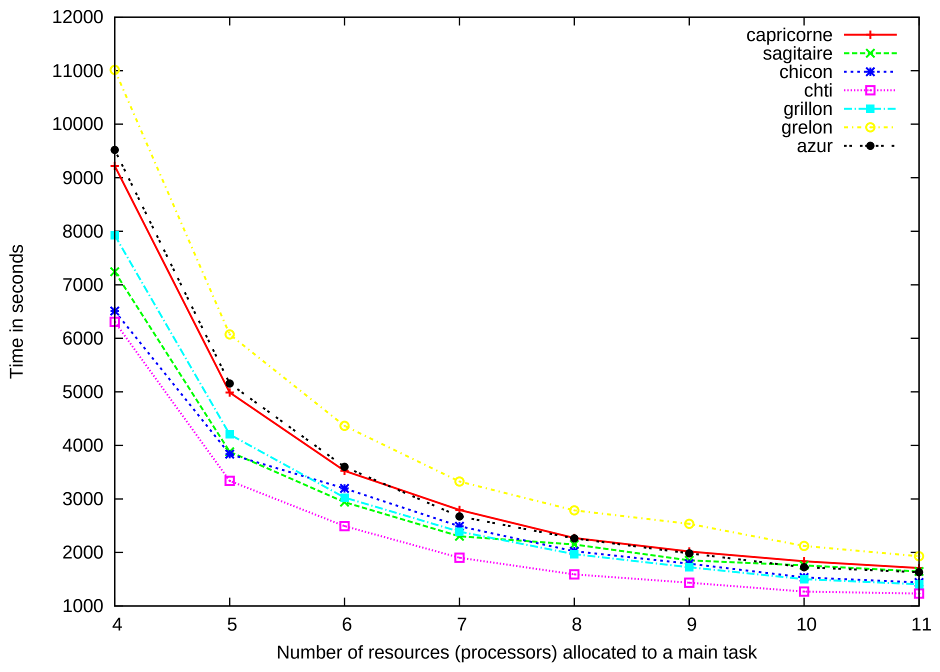

Figure 4: Time needed to execute a main-task on different clusters of Grid'5000.

resources becomes idle, we will schedule the next task of the less advanced DAG on this group. Figure 5 shows the schedule to obtain fairness with 5 scenarios of 3 months. The numbers in the tasks represent the scenario and the current month. Hence, 23 stands for the third month of the second scenario. Since it is a homogeneous platform, scheduling the less advanced task is the same as a Round Robin. In our case, we are going to test all the possible groupings of resources, compute the makespan for each grouping and then choose the best one.

To compute the grouping of resources, we use the same notations as defined in [2]. We recall them for a better understanding of the remaining of the paper:

NS - number of independent scenarios;

NM - number of months in each scenarios;

R - total number of processors;

R_1 - number of processors (among the total R processors) allocated to the multiprocessor tasks;

R_2 - number of processors allocated to the post-processing tasks;

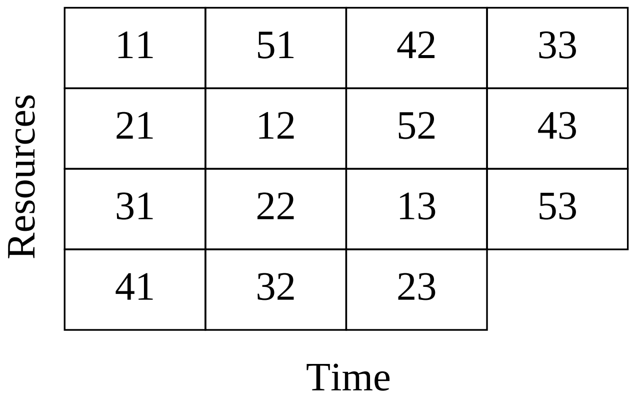

Figure 5: Main-tasks schedule with fairness.

nb_{max} - maximum number of multiprocessor tasks that can run simultaneously given the current choice for the number of processors allocated to a multiprocessor task;

G - number of processors allocated to a single multiprocessor task;

T_G - execution time of a multiprocessor task on G processors;

T_P - execution time for a post-processing task.

Other notations are defined in order to ease the understanding of the formulæ:

nb_{tasks} - number of each type of task ($nb_{tasks} = NS \times NM$);

nb_{used} - number of groups used on the last iteration of the main-processing tasks ($nb_{used} = nb_{tasks} \bmod nb_{max}$).

Since we can not process more than NS simulations simultaneously, we have nb_{max} = min $\{NS, \lfloor R/G \rfloor\}$. Resources allocated to multiprocessor tasks is then $R_1 = nb_{max} \times G$. The remaining resources are allocated to post-processing tasks, so we have $R_2 = R - R_1$.

There are 2 cases to be considered: $R_2 = 0$ (no resource allocated for post-processing during the main-tasks execution) and $R_2 \neq 0$ respectively.

Case 1. $R_2 = 0$**;**

In this case, all the multiprocessor tasks are executed first, followed by the post-processing tasks. The makespan of the multiprocessor tasks is given by:

$$MS_{multi} = \left\lceil \frac{nb_{tasks}}{nb_{max}} \right\rceil \times T_G; \qquad (1)$$

If $nb_{used} = 0$, meaning there are no post-processing tasks executed during the last iteration executing the main-tasks, the total makespan is given by:

$$MS = \frac{nb_{tasks}}{nb_{max}} \times T_G + \left\lceil \frac{nb_{tasks}}{R} \right\rceil \times T_P; \tag{2}$$

Hatched rectangles in Figure 6 represent multiprocessor tasks and light empty rectangles represent the corresponding post-processing tasks.

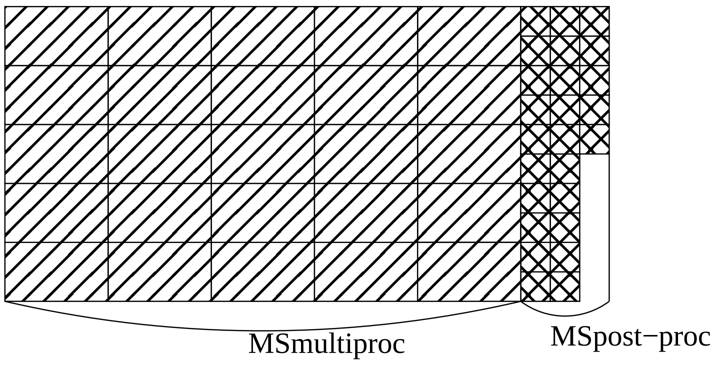

Figure 6: Makespan without processors allocated to the post-processing tasks.

If $nb_{used} \neq 0$, a total of $remPost$ post-processing tasks do not fit on the resources left unoccupied on the last set of multiprocessor tasks ($R_{left} = R - nb_{used} \times G$). With the nb_{used} post-processing tasks corresponding to the last multiprocessor tasks, $remPost$ finally is: $remPost = nb_{used} + max\{0, nb_{tasks} - nb_{used} - \lfloor T_G/T_P \rfloor \times R_{left}\}$;

The makespan (MS) in this situation is:

$$MS = \frac{nb_{tasks}}{nb_{max}} \times T_G + \left\lceil \frac{remPost}{R} \right\rceil \times T_P; \tag{3}$$

Figure 7 shows the resources left unoccupied during the last iteration of the algorithm. Post-processing tasks are scheduled on the newly idle resources.

The case where all post-processing are done at the end is quite simple. We are now going to see how to compute the makespan when post-processing and main-tasks are executed concurrently.

Case 2. $R_2 \neq 0$**;**

In this case, the makespan of the multiprocessor tasks is the same as in Equation (1).

For a set of nb_{max} multiprocessor tasks, the execution time of the corresponding post-processing tasks is given by: $MS_{postproc\_phase} = \lceil nb_{max}/R_2 \rceil \times T_P$.

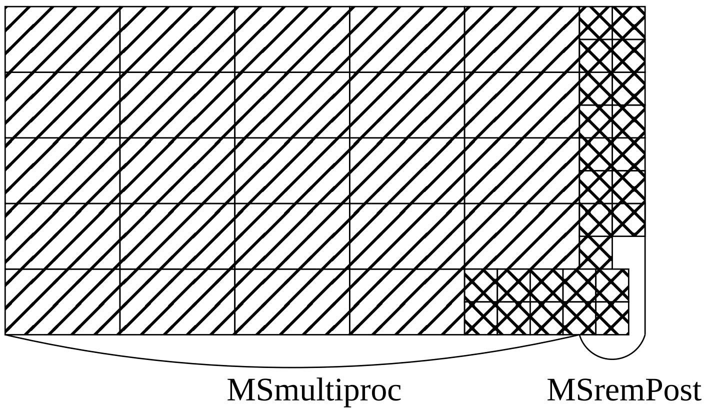

Figure 7: Resources allocation when executing post-processing during the last set of main-tasks.

This time may be greater than the execution time of a multiprocessor task, in which case the execution time for the post-processing tasks will overpass the execution time of the next set of multiprocessor tasks (Figure 8).

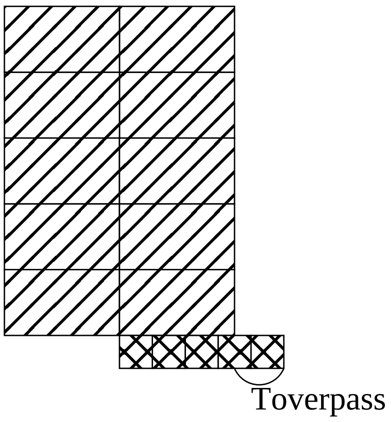

Figure 8: Post-processing tasks overpassing case.

The number of post-processing tasks that can be executed during the interval T_G on the R_2 resources reserved for them is: $N_{possible} = \lfloor T_G/T_P \rfloor \times R_2$.

This value must be tested against the nb_{max} value (since there are nb_{max} multiprocessor tasks generating the same number of post-processing tasks) in order to determine if the R_2

resources left are sufficient or not for the post-processing tasks. If they are not sufficient, the post-processing tasks which do not fit on the resources are reported at the end of the multiprocessor tasks. Otherwise, there may be a part of the R_2 resources which is not used during the whole process. This number of resources left idle is given by: $R_{unused} = R_2 - \left\lceil \frac{nb_{max}}{\lfloor T_G/T_P \rfloor} \right\rceil$.

We denote by n the total number of sets of simultaneous multiprocessor jobs: $n = \lceil nb_{tasks}/nb_{max} \rceil$.

Again, two separate cases must be treated, namely $nb_{used} = 0$ and $nb_{used} \neq 0$.

When $nb_{used} = 0$, the number of tasks reported at the end of the multiprocessor tasks (in the case such tasks exist) is: $N_{overpass} = max\{0, (n-1) \times (nb_{max} - N_{possible})\}$.

In this case, the total makespan is given by:

$$MS = MS_{multi} + \left\lceil \frac{N_{overpass} + nb_{max}}{R} \right\rceil \times T_P \tag{4}$$

In the case $nb_{used} \neq 0$, a total of $N_{overpass}$ post-processing tasks corresponding to the first $n-2$ sets of simultaneous multiprocessor tasks will overpass the execution of the last $n-2$ complete sets of simultaneous tasks (Figure 9): $N_{overpas} = max\{0, (n-2) \times (nb_{max} - N_{possible})\}$

Along with the nb_{max} post-processing tasks from the last complete set of simultaneous multiprocessor tasks, this gives a total of $N_{overtot} = N_{overpass} + nb_{max}$ tasks that should be scheduled starting on the resources left unoccupied in the last set of multiprocessor tasks ($R_{left} = R - G \times nb_{used}$) (Figure 10).

On one processor of the R_{left} remaining ones, $\lfloor T_G/T_P \rfloor$ post-processing tasks can be scheduled. The remaining tasks along with the post-processing task corresponding to the last (incomplete) set of multiprocessor tasks (nb_{used}) is: $remPost = nb_{used} + max\{0, N_{overtot} - (\lfloor T_G/T_P \rfloor \times R_{left})\}$.

Finally, the global makespan when $nb_{used} = 0$ is given by:

$$MS = MS_{multi} + \left\lceil \frac{remPost}{R} \right\rceil \times T_P \tag{5}$$

Using previous formulas, all the 8 possibilities for the parameter G $(4 \rightarrow 11)$ are tested and the one yielding the smallest makespan is chosen. The optimal grouping for various number of resources $(11 \rightarrow 120)$ is plotted in Figure 11. We can see that after 112 resources for 10 scenarios, the grouping is always the same. This is because there are enough resources to have 10 groups of 11 processors, plus one used for the post-processing tasks.

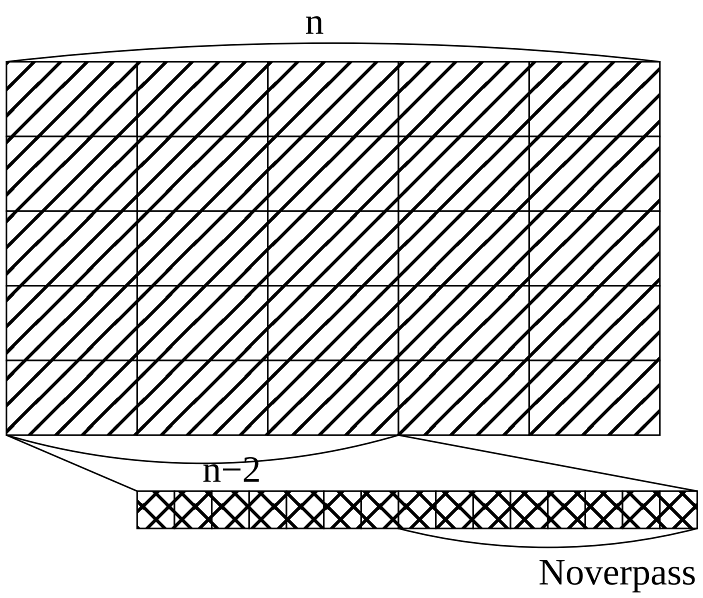

Figure 9: Post-processing tasks overpassing.

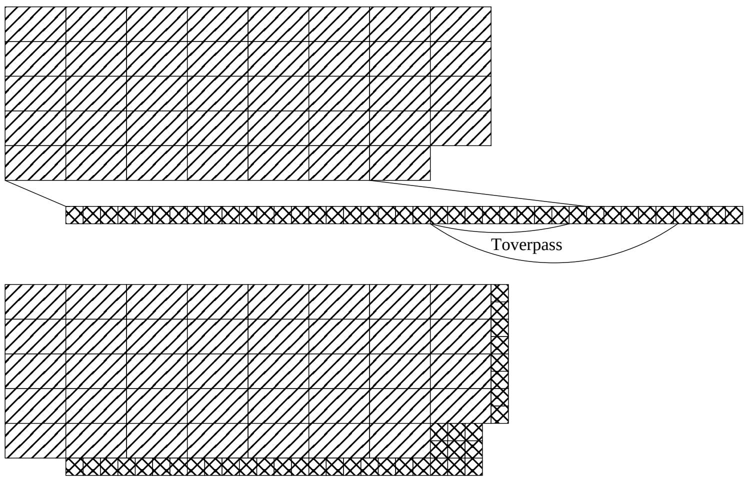

Figure 10: Post-processing tasks overpassing and final schedule.

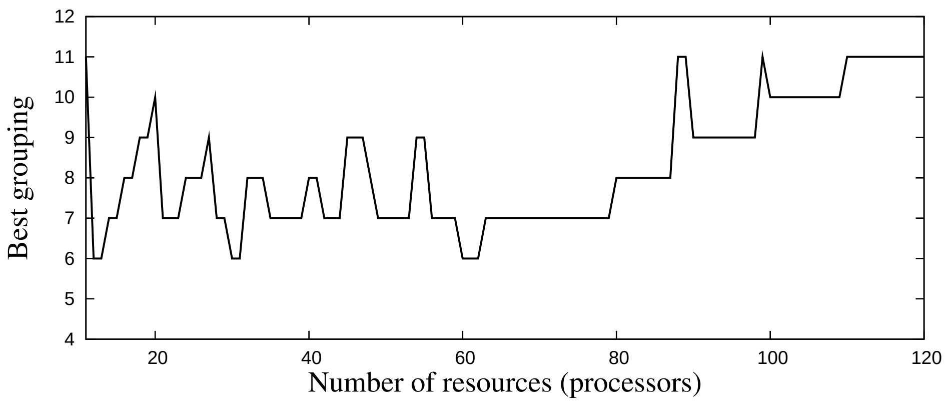

Figure 11: Optimal groupings for 10 scenario simulations.

4.2 Heuristic Improvements

For a given grouping according to the heuristic, it may be possible that, for a set of concurrent multiprocessor tasks and the associated post-processing tasks, all the available resources are not used. *e.g.,* for $R = 53$ resources, and 10 scenarios, the optimal grouping is $G = 7$. Hence a total of 7 multiprocessor tasks can run concurrently, occupying 49 resources. The corresponding post-processing tasks need only 1 resource, which leaves 3 resources unoccupied during the whole computation.

Improvement 1. In order to improve the makespan, the unoccupied resources can be distributed among the others groups of resources. Considering the previous example, we can redistribute the 3 resources left unoccupied among the 7 groups of resources for the multiprocessor tasks. The resulting grouping is 3 groups with 8 resources and 4 groups with 7 resources and 1 resource for the post processing tasks.

Improvement 2. Given that the multiprocessor tasks scale well and that the post-processing tasks have a small duration, another possibility to reduce the makespan is to use the resources normally reserved for post-processing tasks for multiprocessor tasks and to leave all the post-processing at the end. It allows to avoid that the resource used to compute the post-processing become idle while waiting for new tasks to process.

Improvement 3. The previous improvement looks efficient, but it is possible that the best grouping is not a regular one. In some cases, *e.g.,* for 16 resources, it is better to have two groups of 9 and 7 resources, than two groups of 8.

The optimal repartition of the R processors in groups on which the multiprocessor tasks should be executed can be viewed as an instance of the Knapsack problem with an extra constraint (no more than NS simulations can be executed simultaneously). Given a set of items with a cost and a value it is required to determine the number of each item to include in a collection such that the cost is less than some given cost and the total value is as large as possible. Using a Knapsack representation of a problem as been studied in numerous areas such as scheduling [6] and aerodynamics [13].

In our case, there are 8 possible items (groups of 4 to 11 processors). The cost of an item is represented by the number of resources of that grouping. The value of a specific grouping G is given by $1/T[G]$, which represents the fraction of a multiprocessor task being executed during a time unit for that specific group of processors. The total cost is represented by the total number of resources R.

The goal of the division of the processors in groups is to compute the biggest fraction of the multiprocessor tasks during a time interval.

We have n_i unknowns (i from 4 to 11) representing the number of groups with i resources which will be taken in the final solution. Equation (6) has to be maximized under the constraints described in equations (7) and (8).

$$\sum_{i=4}^{11} n_i \times \frac{1}{T[i]} \tag{6}$$

$$\sum_{i=4}^{11} i \times n_i \leq R \tag{7}$$

$$\sum_{i=4}^{11} n_i \leq NS \tag{8}$$

Such a linear program is solved quite fast. The n_i are integers and can only be between 0 and NS. Even with NS really higher than 10 (the number of scenarios we want to schedule), the resolution of the program just takes a few seconds. Furthermore, the grouping given by the linear program is the optimal one for the main tasks, except for the last set of main-tasks.

Simulations comparing the basic heuristic and its optimizations will be presented in Section 5.

4.3 Scheduling on an Heterogeneous Grid

The scheduling heuristics presented in Section 4.1 and 4.2 are designed for homogeneous platforms. We intend to deploy Ocean-Atmosphere on Grid'5000 so we have to adapt the algorithm to be able to work on heterogeneous platforms.

In order to reduce the computation time of NS scenarios, the best way is to divide the set of scenarios into subsets and execute each subset on a different cluster. The choice of the subsets to execute on each cluster is given by Algorithm 1.

Algorithm 1 describes the way the distribution of scenarios is done between clusters. Input parameters are: n, the number of clusters, and “performance” an array. This array has been initialized by the SeD (running on each cluster) using the performance evaluation. The performance evaluation fills a vector with the makespan necessary to execute from 1 to NS scenarios. To know the time needed by cluster C_i for X scenarios, we just have to read the value in $performance[i, X]$. The algorithm behaves as follows: first, the number of scenarios attributed to each cluster is set to 0. Then, each scenario is scheduled on the cluster on which the total makespan increases the less. When all the scenarios are scheduled, this scheduling is returned. This algorithm is realistic because the number of simulations (NS) and clusters (n) are quite low in our case. The number of clusters on Grid’5000 is low, and the number of simulations will be around 10.

Algorithm 1 DAGs repartition on several clusters.

```
for $i = 1$ to $n$ do
    $nbDags[i] = 0$
for $dag = 1$ to $NS$ do
    $MSmin = +\infty$
    $clusterMin = 1$
    for $i = 1$ to $n$ do
        $temp = performance[i][nbDags[i] + 1]$
        if $temp < MSmin$ then
            $MSmin = temp$
            $clusterMin = i$
    $nbDags[clusterMin] = nbDags[clusterMin] + 1$
    $repartition[dag] = clusterMin$
Return $repartition$
```

This algorithm is optimal when considering that the scenarios can not be executed on more than a single cluster. We could launch a scenario on a cluster and continue it on another. We do not use that solution for several reasons. First, the data transmission between two clusters is time consuming, and we should add a lot of calculations to have the optimal algorithm. Its execution time would become really long if computing all the possible data transmissions. Secondly, the algorithm is executed before the execution of the scenarios. This means that if a data transfer is scheduled, it is possible that in reality, the newly selected cluster might not have finished its work. In such a case, the execution time could become far greater than expected.

If the number of scenarios or the number of clusters becomes really large, the "performance" array may become very large. The size of the matrix is $n \times NS$. If the client has not enough memory, the algorithm can not run. Another drawback of this algorithm, with a load increase, is that each cluster must compute estimations of the makespan NS times before the algorithm can start. This estimation could take some time. If this case occurs, the repartition should be done by another heuristic. This did not occur during our experiments, so we did not investigate this problem further.

Since a main task cannot be executed on more than 11 resources, it is possible to compute the number of scenarios that can be assigned on a cluster without deteriorating the makespan. *e.g.,* if we have 40 resources, scheduling 1, 2, or 3 scenarios at the same time will not change anything. 33 resources will be used to execute the main tasks and the overall makespan will stay the same.

Let C_i be a cluster on which we are currently working, R_i the number of resources on this cluster, $T_{post}(1, C_i)$ the time needed to execute one post-processing task on one processor on the cluster and $T_{main}(11, C_i)$ the time needed to execute a main-task on 11 processor on the cluster. To compute how many scenarios can be scheduled on the cluster without degrading the makespan, $nbDags$ must be maximized under the constraints:

$$\underbrace{11 \times nbDags}_{\text{main}} + \underbrace{\left\lceil \frac{T_{post}(1, C_i) \times nbDags}{T_{main}(11, C_i)} \right\rceil}_{\text{post}} \leq R_i \tag{9}$$

$$nbDags \leq NS \tag{10}$$

With the knowledge of the number of scenarios that are not going to slow down the execution, it is possible to schedule this number directly on a cluster instead of one by one, and for the remaining scenarios, use the same technique as described earlier. Doing so diminishes the load of the agent calculating the repartition by reducing the number of iterations made by the algorithm. $nbDags$ can also be used on each cluster to know how many simulations it can compute without deteriorating the makespan. This would allow to execute the heuristic just once for $1 \rightarrow nbDags$ instead of $nbDags$ times when computing the performance estimation.

On heterogeneous platforms, the scheduling is done at two different levels. At a local level (the cluster), the resources grouping is chosen, and at a global level (the grid), the number of scenarios to send to each cluster is chosen. If the clusters are all the same, the algorithm will behave the same as a simple Round Robin to choose the clusters. The local level scheduling is done by the SeD providing the Ocean-Atmosphere service. The global level scheduling is done in the client. It could have been done in the MA, but we do not want to overload it. Its role is to communicate with clients to research services, so it has to be as fast as possible.

The different steps of the execution on several clusters are displayed in Figure 12. (1) the client sends a request to the Master Agent to find the appropriate servers; (2) the MA looks for available SeDs through the DIET hierarchy; (3) each SeD computes an estimation vector containing the time needed to compute from one to NS simulations using the Knapsack representation given in Section 4.2; (4) the performance vector is send back to the MA through the hierarchy; (5) the client receives the estimation vectors; (6) it computes the repartition of the scenarios among the clusters; (7) the client sends multiple requests to the SeDs; (8) Finally, each SeD computes the scenarios it has been assigned.

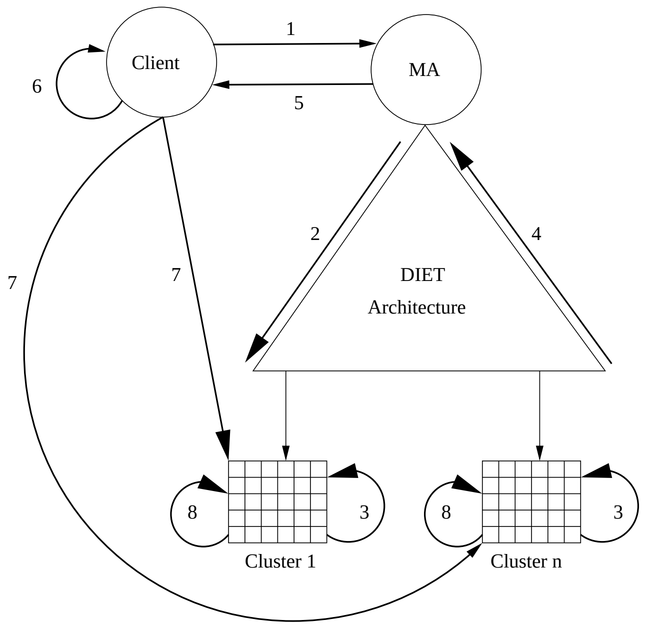

Figure 12: Execution steps to execute the application within DIET.

5 Simulations

To test the heuristics given in Section 4, we performed simulations to analyze their efficiency. First, we estimated the performance on homogeneous platforms, and then on heterogeneous ones.

In this section, the comparisons of makespans are done by simulating a real execution. Performance used to compute the makespans come from benchmarks performed on the different clusters of Grid'5000 (see Figure 4).

5.1 Ocean-Atmosphere Execution

In Section 4.2, we gave 3 improvements of the first heuristic. Gains on the makespan obtained with these improvements presented with respect to the first version of scheduling are plotted in Figure 13. These results come from 5 simulations done on 5 clusters with different computing powers (see Table 1). The figure shows the average of the gains and the standard deviation.

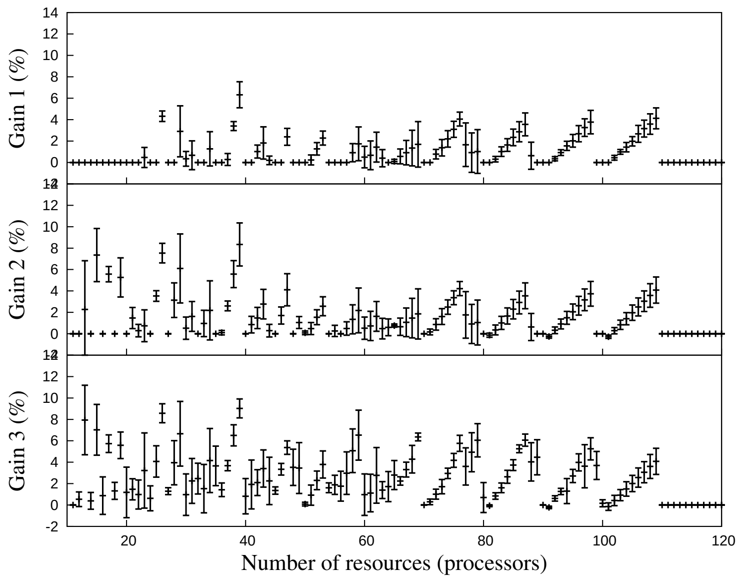

Figure 13: Gains obtained by using resources left unoccupied (Gain 1), using all resources for post-processing tasks (Gain 2), and using the Knapsack problem representation (Gain 3).

The representation as an instance of the Knapsack problem yields to the bests results with low resources. In such a case, it shows consequent gains compared to the other heuristics. When the number of resources grows, the gain decreases, but it is still the best (except in some rare cases). Finally, when there are enough resources, all improvements behave the same as the basic heuristic. Another characteristic shown by this figure is that Gain2 and 3 have a high standard deviations compared to Gain 1. This means that these optimizations are more sensitive to the clusters performances. On average, the Knapsack representation is better than all the others when considering the execution time.

Another interesting comparison is the impact of new resources on the behavior of the heuristics. Figure 14 plots the consequence on the execution time when adding resources. With the basic heuristic, adding resources does not always have the same consequences. When passing from 26 to 27, their is a consequent decrease of the execution time, but from 28 to 29, the execution time does not changes. When redistributing the idle resources to execute the main-tasks, each addition of processor increases the makespan, but not uniformly. Giving all resources to main-tasks decreases the makespan in a more regular manner. Finally, the knapsack representation behaves almost as previously, but the curve is even smoother, meaning that each addition of resource is more taken into account by this heuristic.

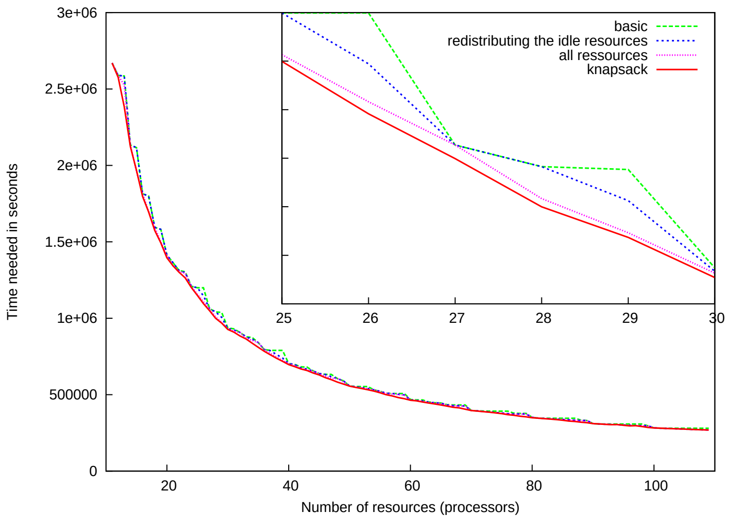

Figure 14: Impact on execution time when adding resources.

The Knapsack representation yields to the best results considering the execution time and the impact of the number of resources. For these two reasons, we choose to implement it to perform the real experiments. To have a better idea of the behavior of this heuristic, Figure 15 shows the different gains obtained in comparison with the basic heuristic. This simulation has been done with the same 5 clusters used in the previous simulation. Between the maximum and the minimum gains, there are a lot of differences. It means that the gain depends of the cluster performance. The diminution of gain occurs because the basic heuristic behaves better. Indeed, it will make the same grouping on each cluster since it divides resources without regarding performance. Thus, in some cases, it will make a grouping close to the one found by the Knapsack representation. We can note that the Knapsack representation varies a lot depending on the clusters, but it almost never behaves worse than the basic heuristic. When it does behaves worse, it is less than 0.5%, which is acceptable. The large differences concord with the high standard deviation presented in Figure 13.

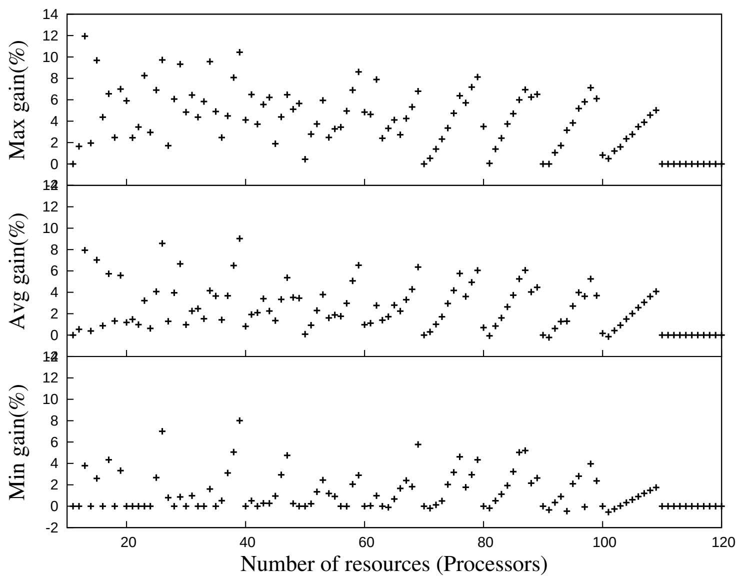

Figure 15: Gains obtained with the Knapsack representation (Min, Average and Max).

Figure 16 shows the gains obtained by the Knapsack representation compared to the basic heuristic on several clusters. Clusters have all the same number of resources. The simulation was done with 10 scenarios. The x-axis represents the number of clusters and the number of resources per cluster. Hence, 2.25 represents the results for two clusters with 25 resources each. To map scenarios on clusters, we used the repartition algorithm described in Section 4.3. This simulation shows that it is possible to attain a 12% gain, but in most cases, the gains are between 0 and 8. Using the Knapsack representation, the grouping of processors is different from the grouping with the basic heuristic. Therefore the makespan will be different on each cluster, so the repartition of the scenario may be different. We can note that with several clusters, there are phases when there are no gains at all. This is due to the fact that the slowest cluster has enough resources to execute the scenarios it has been assigned so both heuristics produce the same grouping. While the overall makespan is the same with both techniques, the other clusters may have a different grouping. So, even if there is no makespan improvement, the other resources usage is improved.

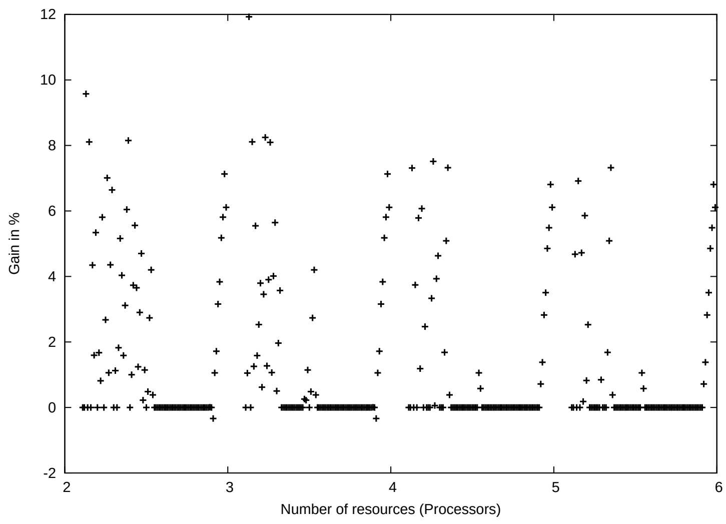

Figure 16: Gains obtained by Knapsack on several clusters (51.01% of executions benefit of Knapsack).

5.2 Repartition on Clusters

To analyze the repartition algorithm, we used the execution time of benchmarks made on 5 clusters of Grid'5000. Each cluster is given the same number of processors.

We have compared the repartition done on the clusters using Algorithm 1, presented in Section 4.3, to a simple Round Robin. Figure 17 presents the comparison of the two algorithms with 10 scenarios of 180 months each. All the values for the resources from 11 to 120 are used for this comparison. On the clusters, in both cases, the heuristic used to group resources is the one using the Knapsack representation.

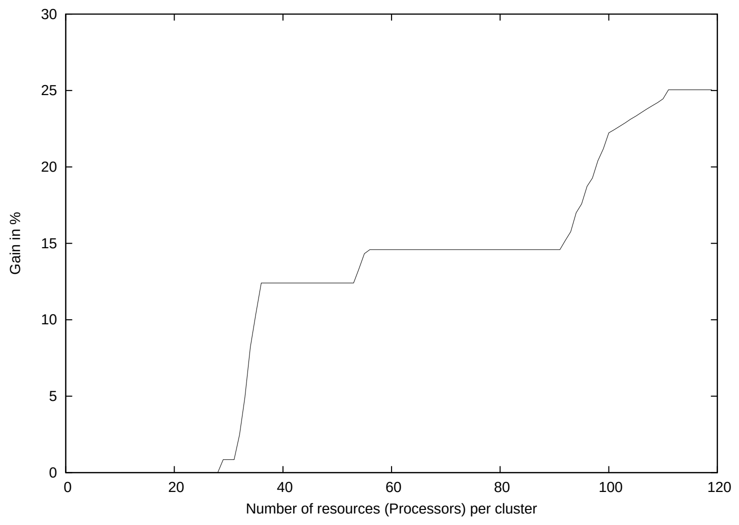

Figure 17: The repartition heuristic compared to a Round Robin.

We can see that gains are more and more important when more resources are available. Another point is that there are some steps. In such a case, the same cluster stays the slowest for some time. The gain improves later when it becomes better to map one of the scenarios on a faster cluster. When reaching 112 resources, the gain stays constant. In the example, the gain is around 25%. This number corresponds exactly to the difference of execution time between the fastest and the slowest cluster when computing one monthly simulation on 11 processors. The fact that it is the difference between times for one monthly simulation is normal because with a lot resources, the execution time of 1 or 10 scenarios is the same.

Figure 18 plots the comparison of the two algorithms, but with a bigger load. The resources are increased by steps of 25. Even with an increase of the load, the behavior stays almost the same. The steps are still present, but the more scenarios, the longer the stagnation. With a lot of resources, all the optimizations reaches also a gain of 25%. The difference with a small load is that with a lot of scenarios and few resources, the gain is not 0% since changing the repartition will change the makespan.

It is possible to predict the behavior of the repartition algorithm. Since the gains depends of the clusters performance, when clusters are almost homogeneous, the algorithm will behave almost as a Round Robin. Reciprocally, the more heterogeneous the clusters are, the more the gain will be elevated.

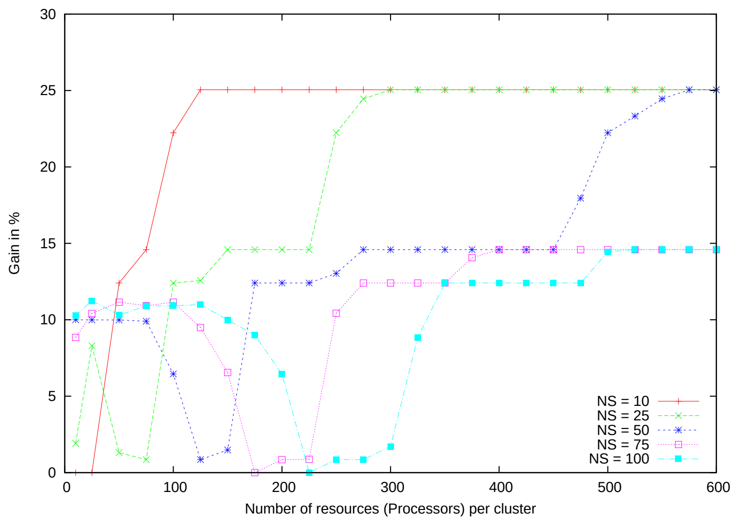

Figure 18: The repartition heuristic compared to a Round Robin with lots of scenarios.

Figure 19 shows the execution time of the experiment with both algorithms. Until 29 resources, the time is the same, which concord with the 0% gain in Figure 17. We also see the steps in the execution time of the repartition algorithm.

Figure 20 presents the gains in terms of time between the two algorithms with a high number of scenarios. With 10 resources per cluster and 100 scenarios, the gain is more than a week (≈655000 seconds).

Previous tests shown that the repartition algorithm is quite efficient, and it will not have to make too much calculations. So, we will use the Knapsack representation as well as the repartition algorithm to perform the real experiments.

6 Experimental Results

In order to perform real experiments, we had to implement some new features within DIET. The schedulers implemented within DIET are local ones, meaning that the SeDs are sorted at each step of the hierarchy, using the estimation vector, and the list is returned to the client which takes the first one. The repartition algorithm needs the knowledge of the performances of all the SeDs. The whole set servers are known only at the top of the hierarchy by the MA and the client. In order not to disturb the services research done by the MA, we

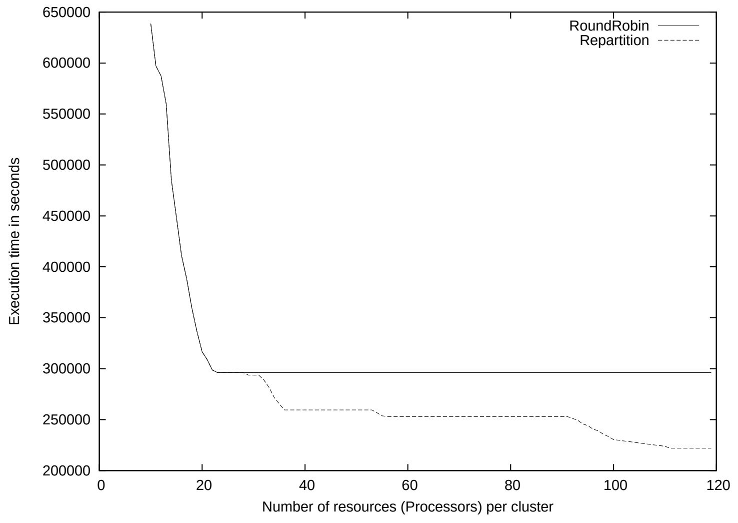

Figure 19: The repartition heuristic compared to a Round Robin.

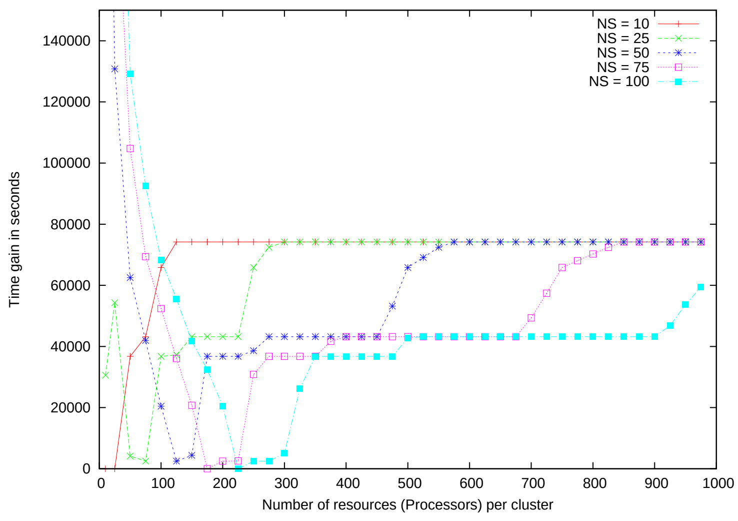

Figure 20: The repartition heuristic compared to a Round Robin.

decided to implement the repartition algorithm within the client. This avoids to overload the agent.

The first feature to implement was the call to multiple servers. Each server must execute a subset of the scenarios and execute it. To execute a request, the client has to use `diet_async_call()` to send the request to a cluster. The problem is that this method can only send requests to one cluster at a time. So, the first new feature that had to be implemented was the possibility to make submissions to several clusters with just one `diet_async_call()` from the client. We decided to modify the API. This could have been avoided, but we wanted the repartition to be done on the API level, not by the programmer of the client. Three reasons support this choice. First, if the client programmer does it, he has to access the DIET internal datas, but he is not supposed to. Secondly, if it is in the API, the code can be re-used to implement another global scheduling algorithm for another application. Finally, doing it at the API level avoids the modification of the client code. Hence, to distribute DAGs among servers, we added a call to the repartition algorithm into `diet_async_call()`. Once the scheduling is done, the remaining code of the function is executed several times to send each subset of scenarios, one subset for each selected cluster. This works fine, but we loose some compatibility with existing tools working with DIET. Each request should have a unique `requestID`, but the attribution of the `requestID` is done before the execution. So, for each execution requested on a cluster, the ID is the same. *e.g.,* VizDiet is a tool used to monitor the requests, so it is not supposed to have several requests with the same ID. We intend to develop a generic way to add a global scheduling algorithm in the client and to modify the code to keep the compatibility with all existing tools. Another consequence of the multiple calls is the need to wait for all the executions to be done before terminating the client which was also implemented.

Another point that had to be taken into account to perform real experiments is the addition of some fault tolerance in the SeDs. When there is a problem during the execution, it must be possible to restart a scenario from the last month that has been properly executed. The application freezes on some occasions which are still unknown and relaunching the scenario, starting from the last month correctly executed, allows a proper execution. Another reason for this feature is that Grid'5000 is not a production grid. Each person can only reserve nodes for some time. In our case, the reservation is limited to 10 hours. Hence, executing an entire experiment of 10 scenarios each with 1800 months is impossible. The place is really limited so we have to relaunch the experiment many times to complete it. We launched an entire experiment on 5 clusters. If we could have all the resources needed and all the time, the execution would last about one month and a half. With the resources number and the reservations on Grid'5000, it took us more than 5 months to execute an entire experiment. On a production platform, we would not have this problem.

As presented in Section 4.2, the Knapsack representation leaves the post-processing tasks to the end of the execution. The data exchanges between each months being around 1 GB, we also had to think of a way to flush the data. On Grid'5000, the datas are stored on an NFS. The total space of the NFS on the clusters we are working on is 500 GB per cluster. So, considering the high number of users, it is really easy to fill it up entirely. To avoid that, we can flush the datas by executing the post-processing tasks. This should not slow the execution too much. The maximum delay that can be produced is $T_P \times nbFlushs$. Indeed, all resources will be used to execute the post-processing tasks, except when there are not enough remaining tasks. Without flush, this occurs just once at the end of the execution, but in this case, it occurs at each flush. Figure 21 presents a possible execution with flush in the middle. This feature is not yet implemented, but it can be done quickly, using Cori, a tool interacting with DIET used to obtain system metrics. So, we could say that when there is less than a given amount of GB left on the NFS, a flush is necessary. Estimate this disk space needed can be quite hard, because the NFS is shared among different users, so it is possible that when checking the available disk space, there is enough, but during the execution of a main task, another user fills it up. In such a case, we will have to perform a flush, and then re-execute the main-tasks where the space was not sufficient. If this occurs, the slowdown will be higher than expected. The main point here is to find a compromise between the numbers of flush and when to perform them. *e.g.*, we could flush when the available storage space is inferior to $2 \times nb_{max}$ which leaves some error margin.

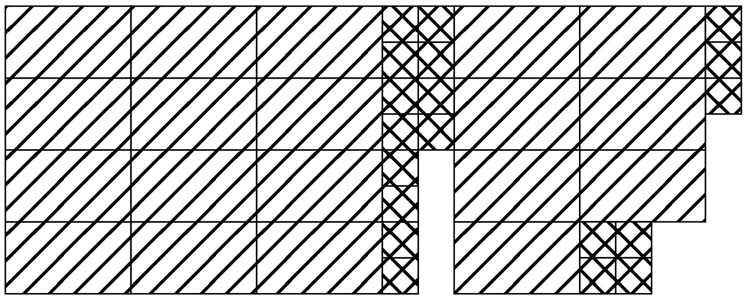

Figure 21: Execution with flush.

Another problem caused by the large data produced by the application is the execution of the post-processing tasks. Let us take an example experiment composed of 5 scenarios of 50 months each executed on a single cluster with 50 processors. The post-processing tasks will be executed each on 1 processor. Thus, 50 tasks will be executed concurrently. The normal execution time for a single task is between one and two minutes. But, because each task will have to transfer the data produced by the corresponding main-task, around 50 GB ($50 \times 1GB$) must be transferred at the same time. All data are stored on the same NFS, so the execution time will be slowed by the NFS speed or the network bandwidth. In some cases during our tests, an execution of concurrent tasks took more than 30 minutes to execute instead of 2. Considering the flush problem and the execution of concurrent post-processing tasks, it is better to have some resources to execute them during the main-tasks execution. This will deteriorate the theoretical makespan, but in practice, it may often be a better solution. An easy way to implement this is that we can keep a node on each cluster that will be dedicated to the execution of post-processing tasks.

The last problem we had when preparing the real experiments, was a bug due to MPI. If there are 2 scenarios executing on the same node, the application behaves abnormally and both scenarios crash. The MPI used by Ocean-Atmosphere is OpenMPI, but it is going to be replaced by MadMPI[3] [15]. When the replacement of MPI will be done the problem might be solved, but we are currently investigating this problem because it can occur quite often. All nodes on Grid'5000 are at least bi-cores, and there are clusters with bi-processors each with 2 cores. To avoid the problem, we added another constraint to the linear problem solved to obtain the grouping. The new constraint forces the number of processors in each group to be divisible by the number of cores of the nodes (2 or 4). Thus, the makespan changes since the grouping does. When working on clusters with 4 cores per node, we cannot use more than 8 processors for a single main-task because it needs between 4 and 11 processors. To allow the use of bigger groups of resources, we added the execution time on 12 processors, with the same value as the execution on 11. This avoids a big slowdown of the application when working on a lot of resources. More processors are used, but the execution time is shorter. Figure 22 plots the average loss of time on 5 different clusters between the 2 extra-constrained Knapsack representation (with and without adding the 12^{th} processor) in regards to the original version. With a few resources, both versions are bad. But, when the number of resources grows, the version with the 12^{th} processor behaves better. With many resources, this new version gives the same results as the original one. Instead of 112 resources to obtain the best makespan, the Knapsack version using 12 processors as maximum number of processors needs 121 processors (10 groups of 12 resources and 1 to execute the post-processing tasks).

To test the accuracy of our simulations, we compared the simulated times of several experiments with the real execution times. We did not used any resource to execute post-processing tasks during the main-tasks phase. Figure 23 plots the differences between simulations and experiments. The positive difference means that the real experiment is slower than the simulation, and the negative means that is it faster. Experiments have been con-

[3] http://runtime.futurs.inria.fr/MadMPI/

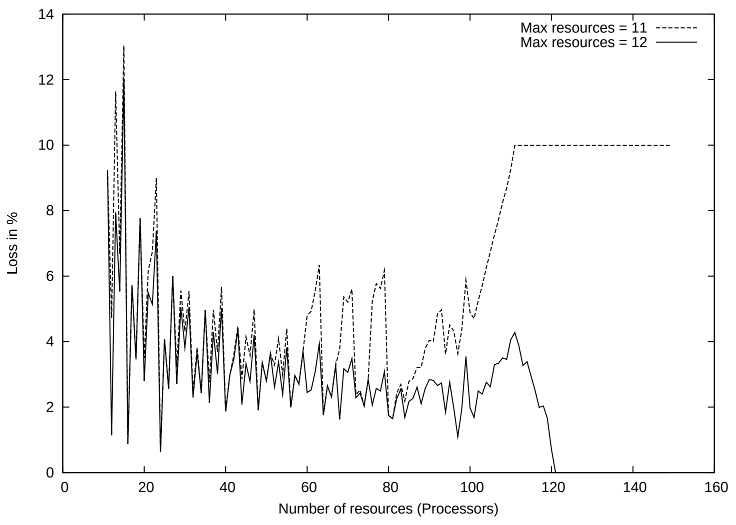

Figure 22: Comparison between 3 versions of the Knapsack representation.

ducted on small scenarios. We did it because we did not wanted to perturb the results with the restart phases and not to fill up the NFS since the flush is not yet implemented.

The difference between the simulations and the real execution time is sometimes large. Concerning the differences between the execution time of the main-tasks and the simulated times, it is not so bad except in the fourth experiment. Most of the time, The real computation time is less than 7% slower than the simulated time. But in the fourth experience, it is 14% faster. In the seventh experiment, the real execution is also 7.8% faster. The average difference between simulations and the real experiments is 6.3%. This difference is quite good, but as we can see the results are very different from one experiment to another.

These results show that simulations really depend on the cluster load. The benchmarks were made with a specific cluster load, so if this load changes during the execution, the time will be different from the one expected.

The slowdown due to the post-processing tasks is also well represented by this figure. In all experiments, the main-tasks time is quite good, but the total time is bad except in the third one. So, the post-processing tasks are really slowing the execution. The third experiment is good because the post-processing tasks where executed by groups of 4. When executed by groups of 4, the NFS used for the experiment is able to have the same performances that with only one post-processing task.

The average difference between simulations and real experiments is 20.8% for the total execution time. This difference is big and still has to be improved to allow the simulations to be as precise as possible.

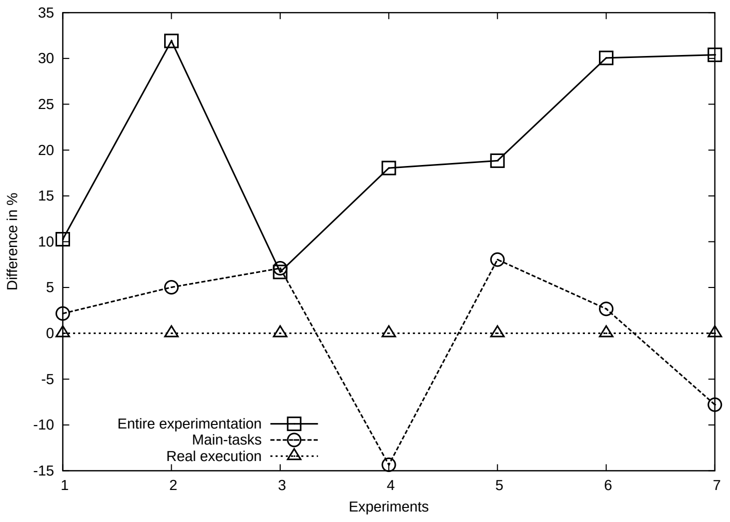

Figure 23: Comparison between simulation and real experiments.

7 Conclusion and Future Work

World's climate is currently changing. In order to predict its variations, simulations are conducted by climatologists. Computation of such simulations is very time consuming so it has to be optimized. This paper presents the work of analyzing and modeling a real climatology application (Ocean-Atmosphere) with the purpose of deriving appropriate scheduling heuristics in order to obtain good completion times for a real implementation over the grid.

First, the computation needs have been modeled as independent identical 1D-meshes derived through the chaining of several basic DAGs. Then a simplified model with clustered tasks based upon the actual time parameters of the application has been derived.

For this new model, a first scheduling heuristic has been designed. The basic idea is the allocation of the same number of processors to all multiprocessor tasks and leaving what is left to post-processing tasks. Three improved versions have been proposed. The best improved version is the one that models the resources allocation as an instance of the

Knapsack problem with a supplementary constraint. The three improved versions have been simulated and yielded to gains of up to 12%.

Then, this method was adapted to be applicable on a heterogeneous grid composed of homogeneous clusters. Distributing the simulations among different clusters reduces the overall makespan of the application. The faster a cluster is, the more scenarios it will have to execute.

When going from theory to practice, several problems occur. Some due to bugs in libraries we use and some from hardware performance. To be able to perform a real experiments, we had to modify the heuristic which yields to loss of performance. We also had to add more features to tackle the storage of huge data sets.

The application and the scheduling heuristic have been implemented in DIET. When comparing the real execution times with the simulated times, some big differences arise in some cases because of the post-processing tasks. The NFS performances are the origin of this problem. Another factor influencing the differences between real times and simulated times is that execution times have been benchmarked, so if the load of a cluster changes, the execution time is not stable.

The correction of the bugs, and the diminution of performance when performing real experiments are the weaknesses we are currently trying to overcome. We also want to change the implementation of the repartition between clusters to be able to use a inherited method in the client to implement other algorithms.

Future work also consists in extending the present work to a generic heuristic that can schedule the same kind of workflow, made of independent chains of identical DAGs composed of moldable tasks.

Another possible direction to continue this work is the integration of the presented algorithms with a specialized DIET agent called the MaDAG. This agent is designed to schedule workflows of services in a DIET architecture.

Finally, we would like to make use of Simbatch (a batch system simulator based on Simgrid) to be able to predict when resources will be free, and so, be able to make resources reservation automatically, while keeping the good performances.

References

[1] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, El-G. Talbi, and I. Touché. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[2] Y. Caniou, E. Caron, G. Charrier, A. Chis, F. Desprez, and E. Maisonnave. Ocean-atmosphere modelization over the grid. In *The 37th International Conference on Parallel Processing (ICPP 2008)*, 2008.

[3] E. Caron and F. Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[4] M. Deque, C. Dreveton, A. Braun, and D. Cariolle. The ARPEGE/IFS atmosphere model: a contribution to the french community climate modeling. *Clim Dyn*, 10:249–266, 1994.

[5] C.G. Knight, S.H.E. Knight, N. Massey, T. Aina, C. Christensen, D.J. Frame, J.A. Kettleborough, A. Martin, S. Pascoe, B. Sanderson, D.A. Stainforth, and M.R. Allen. Association of parameter, software and hardware variation with large scale behavior across 57,000 climate models. *Proceedings of the National Academy of Sciences*, 104:12259–12264, 2007.

[6] C. Lu and S.M. Lau. An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes. *International Conference on Distributed Computing Systems*, pages 629–636, 1996.

[7] G. Madec. *NEMO Reference manual, ocean dynamic component: NEMO-OPA*. Number 27. Institut Pierre Simon Laplace (IPSL), 2006. ISSN 1288-1619.

[8] T. Oki and Y. C. Sud. Design of Total Runoff Integrating Pathways (TRIP)-A Global River Channel Network. *Earth Integration*, 2(1234):1–37, 1998.

[9] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, page 39. IEEE Computer Society, 2001.

[10] A. Radulescu and A. J. C. van Gemund. Low-cost mixed task and data parallel scheduling. In *30-th International Conference on Parallel Processing (ICPP)*, pages 69–76, August 2001.

[11] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, November 1997.

[12] T. Rauber and G. Runger. Compiler support for task scheduling in hierarchical execution models. *Journal of Systems Architecture*, 45(6-7):483–503, 1999.

[13] J.M. Romaine. *Solving the Multidimensional Multiple Knapsack Problem with Packing constraints using Tabu Search*. PhD thesis, 1999.

[14] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *Journal of Parallel and Distributed Computing*, 60(3):297–319, 2000.

[15] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A multithreaded communication engine for multicore architectures. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, April 2008. IEEE.

[16] S. Valcke, R. Caubel, A. Vogelsang, and D. Declat. Oasis 3, user guide. Technical Report PRISM Report Serie no 2 (5th edition), CERFACS, Toulouse, 2004.

[17] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2006.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399