Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations

Tchimou N'Takpé, Frédéric Suter

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations

Tchimou N'takpé — Frédéric Suter

N° 6774

Décembre 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6774--FR+ENG

Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations

Tchimou N'takpé*, Frédéric Suter†

Thème NUM — Systèmes numériques
Équipe-Projet ALGORILLE

Rapport de recherche n° 6774 — Décembre 2008 — 19 pages

Abstract: Scheduling multiple applications on heterogeneous multi-clusters is challenging as the different applications have to compete to access the resources. A scheduler thus has to ensure a fair distribution of the resources among the applications and prevent harmful selfish behaviors while still trying to minimize their respective completion time. In this study we consider mixed-parallel applications, represented by graphs whose nodes are data-parallel tasks, that are scheduled in two steps: allocation and mapping. We investigate several strategies to constrain the amount of resources the scheduler can allocate to each submitted application. This study is then evaluated over a wide range of scenarios.

Key-words: Concurrent scheduling, mixed parallel application, DAG, parallel task, multi-cluster platform

* Nancy University / LORIA – UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1 – Tchimou.Ntakpe@loria.fr
† CC-IN2P3 / CNRS – USR 6402 CNRS – Frederic.Suter@cc.in2p3.fr

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Ordonnancement concurrent de graphes de tâches parallèles sur plates-formes multi-grappes avec des allocations sous contraintes d'utilisation de resources

Résumé : Ordonnancer plusieurs applications sur plates-formes hétérogènes multi-grappes est difficile du fait que différentes applications sont en compétition pour l'accès aux ressources. Un tel ordonnancement doit assurer une distribution équitable des ressources entre les applications et éviter des comportements individualistes nuisibles tout en essayant de minimiser leurs temps de complétions respectifs. Dans cette étude, nous considérons des applications parallèles mixtes dont les nœuds sont des tâches parallèles. L'ordonnancement s'effectue en deux étapes: phase d'allocation et phase de placement. Nous étudions plusieurs stratégies permettant de restreindre la quantité de ressources à allouer à chaque application. Nous évaluons ensuite ces stratégies pour une large gamme de scénarios.

Mots-clés : Ordonnancement concurrent, application parallèle mixte, DAG, tâche parallèle, plate-forme multi-grappes

1 Introduction

Nowadays it is a common thing for institutions such as universities or computer centers to give access to compute platforms comprising several clusters. Such architectures have spread during the last decade in the field of grid computing that made resource federation a popular concept. These multi-cluster platform configurations are often accessed through a resource manager and their compute nodes are shared among the different users of the platform. Each of them can submit one or several applications (or jobs) to the resource manager that is in charge to place each of these applications on a particular set of compute nodes. Moreover such multi-cluster platforms can be heterogeneous, in terms of computing power and network interconnection. As the clusters are generally located in a single site, the network latency between the different nodes is that of a LAN. The produced schedules are thus not impacted by large inter-cluster communications as it would be the case in a more general grid platform connected through a wide area network.

Such a context raises the following question: "how to concurrently schedule multiple applications while minimizing the perturbations between them and getting the best output from the platform?" To address the concurrency issue, several researchers have attempted to design scheduling heuristics in which the task graphs representing the different applications are aggregated into a single graph to come down to the classical problem of scheduling a single application [15], or hierarchical schedulers in which applications are first dispatched among clusters and then relying on waiting queues algorithms [4, 8]. A limitation of these scheduling algorithms is that they assume that the application graphs only comprise sequential tasks. But a way to take a higher benefit from the large computing power offered by multi-clusters is to exploit both *task parallelism* and *data parallelism*. Parallel applications that use both types of parallelism, often called *mixed parallelism*, are structured as *parallel task graphs* (PTGs), *i.e.,* Directed Acyclic Graphs (DAG) whose nodes are data-parallel tasks and edges between nodes represent precedence and/or communication between tasks, (see [3] for a discussion of the benefits of mixed parallelism and for application examples). Several algorithms for the scheduling of PTGs on heterogeneous multi-clusters exist [11] but they consider that all the platform is dedicated to a single application. Consequently these heuristics may produce schedules that require a lot of resources that can negatively impact (or be impacted by) other scheduled applications in a shared environment.

To schedule a PTG, a classical approach is to separate the scheduling process in two steps: one to *allocate* each task, *i.e.,* to determine the number of processors on which execute it, another to *map* these allocated tasks onto the platform using standard list scheduling algorithms. A first study about the concurrent scheduling of multiple PTGs has been presented in [10], focusing on the allocation step. The idea developed in that paper was to ensure a fair sharing of the resources between the concurrent applications by applying a resource constraint when allocating processors to the tasks of each PTG. For instance if ten PTGs have to be scheduled simultaneously, each of them could be constrained to use at most one tenth of the processing power of the platform to build its schedule. Two procedures, called SCRAP and SCRAP-MAX, were presented that guarantee the respect of such a resource constraint but let the questions of the determination of this constraint and of the concurrent mapping

open. Consequently there was no validation of the benefits induced by resource constrained allocations in presence of real concurrency provided in [10].

The contributions of the present work are: (i) to propose a mapping procedure that increases fairness for the second step of a parallel task scheduling heuristic; (ii) to investigate different strategies for the determination of the resource constraint; (iii) to evaluate and compare the different resulting scheduling heuristics, in terms of fairness and average global completion time and (iv) to provide a experimental validation of our approach based on *resource constraints*.

This study is organized as follow. Section 2 presents the platform and application models used in our study. Section 3 discusses related work. Section 4 recalls previous results on constrained allocation procedures. Section 5 discusses how to map several PTGs concurrently. Section 6 presents several strategies to determine a resource constraint that ensures a good fairness while minimizing the total completion time of the scheduled applications. We evaluate the resulting scheduling heuristics in Section 7. Finally, Section 8 concludes the paper with a summary of our findings and perspectives on future directions.

2 Platform and application models

In this study we base our platform model on a real-world multi-cluster platform: Grid'5000[1]. The goal of Grid'5000 is to build a highly reconfigurable, controllable and monitorable platform to allow experimental parallel and distributed computing research. The platform consists of nine geographically distributed sites, aggregating a total of 5,000 CPUs. Each of the nine sites hosts at least one commodity cluster, and the number of processors per cluster ranges from around 100 to around 1,000.

We consider 4 particular sites of Grid'5000 that comprise multiple clusters. Table 1 gives the name of each cluster along with its number of processors and processing speed expressed in GFlop/s. These four sites differ in terms of total number of processors (99, 167, 229 and 180 respectively) and heterogeneity (20.2%, 6.1%, 36.8% and 34.7% respectively). The heterogeneity of a platform is determined by the ratio between the speeds of the fastest and slowest processors. The interconnection network also differs depending on the site as the clusters of Rennes and Lille are connected to the same switch while in Nancy and Sophia, each cluster has its own switch. This leads to different contention conditions in our target platforms.

More formally, each platform consists of c clusters, where cluster $C_k, k = 1, \ldots, c$ contains p_k identical processors. A processor in cluster C_k computes at a speed s_k expressed in flop/s.

A PTG application is modeled as a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \ldots, V\}$ is a set of vertices representing data-parallel tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges between vertices, representing communications between tasks. Each edge $e_{i,j}$ has a weight, which is the amount of data (in bytes) that task v_i must send to task v_j. Note that in addition to data communication itself, there may be an overhead for data redistribution, *e.g.,* when task v_i is executed on a different number of processors than task v_j. Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task. Since data-parallel tasks can be executed on various

[1] `http://www.grid5000.fr`

Site	Cluster	#proc	Gflop/s
Lille	Chuque	53	3.647
	Chti	20	4.311
	Chicon	26	4.384
Nancy	Grillon	47	3.379
	Grelon	120	3.185
Rennes	Parasol	64	3.573
	Paravent	99	3.364
	Paraquad	66	4.603
Sophia	Azur	74	3.258
	Helios	56	3.675
	Sol	50	4.389

Table 1: A selection of multi-cluster subsets of the Grid'5000 platform.

numbers of processors, we denote by $T^k(v,p)$ the execution time of task v if it were to be executed on p processors of cluster C_k. In practice, $T^k(v,p)$ can be measured via benchmarking on each cluster for several values of p, or it can be calculated *via* a performance model. 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.

We take a simple approach for modeling data-parallel tasks. We assume that a task operates on a dataset of d double precision elements (for instance a $\sqrt{d} \times \sqrt{d}$ square matrix). We arbitrarily assume that processors have at most 1GByte of memory and thus $d \leq 121M$. We also assume that d is above $4M$ (if d is too small, the data-parallel task should most likely be fused with its predecessor or successor). The volume of data communicated between two tasks is equal to $8 \times d$ bytes. We model the computational complexity of a task, in number of operations, with one of the three following expressions, which are representative of common applications: $a \cdot d$ (*e.g.,* a stencil computation on a $\sqrt{d} \times \sqrt{d}$ domain), $a \cdot d \log d$ (*e.g.,* sorting an array of d elements), $d^{3/2}$ (*e.g.,* a multiplication of $\sqrt{d} \times \sqrt{d}$ matrices). For the first two types of complexity a is picked randomly between 2^6 and 2^9, to capture the fact that some of these tasks often perform multiple iterations. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three.

While the above provides a model for sequential task execution we also need to account for parallel executions, *i.e.,* for how to determine $T^k(v,p)$ when p varies. We use a simple 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 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 exhibits different execution times for different numbers of processors.

We consider random application graphs that consist of 10, 20, or 50 data-parallel tasks. We use four popular parameters to define the shape of the PTG: width, regularity, density, and "jumps". The width determines the maximum parallelism in the PTG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The

regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the PTG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2, 0.5, and 0.8 for width, and 0.2 and 0.8 for regularity and density. Furthermore we add random "jumps edges" that go from level l to level $l+jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). We refer the reader to our DAG generation program and its documentation for more details [13].

In addition to these synthetic PTGs we also consider real PTGs from the Strassen matrix multiplication and from the Fast Fourier Transform (FFT) application. Both are classical test cases for PTG scheduling algorithms and we refer the reader for instance to [5] for details on their structure. These PTGs are more regular than our synthetic PTGs, which are more representative of workflow applications that compose arbitrary operators in arbitrary ways.

3 Related Work

Several authors have studied the concurrent scheduling of multiple applications onto heterogeneous platforms [4, 8, 15]. Authors of [15] address that issue by combining the different DAGs representing the applications into one single DAG. They propose two scheduling heuristics aiming at minimizing the completion time of the combined DAG while ensuring a fair schedule for each of the original applications. A two-level distributed scheduling algorithm for multiple DAG also has been proposed in [4]. The first level is a WAN-wide distributed scheduler responsible for dispatching the different DAGs (viewed at this level as a single task) to several second level schedulers that are LAN-wide and centralized. The focus of this study is more on environment-related issues, *e.g.,* task arrival and machine failure rates or wait queue sizes, than on scheduling concerns, *e.g.,* ensuring a fair access to the resources. The hierarchical competitive scheduling heuristic for multiple DAGs onto heterogeneous platforms provided in [8] proposes a framework in which each application is responsible of its scheduling, and thus with no direct knowledge of the other applications. All these algorithms or environments focus on DAGs and not PTGs, *i.e.,* on applications only comprising sequential tasks. Consequently the issue of determining on how many processors a task should execute, which is the core of the present work, does not arise in these researches.

On the other hand, two heuristics were recently proposed: HPCA [9] and MHEFT [1] to schedule a single PTG on a heterogeneous platform. HCPA extends the CPA algorithm [12] to heterogeneous platforms by using the concept of a homogeneous reference cluster and by translating allocations on that reference cluster into allocations on actual clusters containing compute nodes of various speeds. MHEFT extends the well-known HEFT algorithm for scheduling DAGs [14] to the case of data-parallel tasks. Weaknesses in both HCPA and MHEFT were identified and remedied in [11], which performs a thorough comparison of both improved algorithms and finds that although no algorithm is overwhelmingly better than the other, HCPA would most likely achieves a cost-effective trade-off between application makespan and parallel efficiency (*i.e.,*

how well resources are utilized). But no investigation were conducted on the behavior of these two heuristics when they have to schedule multiple PTGs simultaneously.

Scheduling a PTG on a (heterogeneous) multi-cluster platform can be related to scheduling a multi-threaded programs on a (heterogeneous) multi-core system as in [6]. When moving to multiple PTG/programs, a main difference raises on how the scheduled entities share the resources. On a multi-cluster platform, two tasks mapped on the same cluster will not share processors whereas two multi-threaded jobs mapped on the same core will share time slices. In the present work we address the fair resource access in a space sharing context.

4 Constrained Resource Allocation

In this section we briefly recall how it was proposed in [10] to constrain the resource amount that can be used during the allocation process to schedule a single PTG. In that work a resource constraint, denoted by β was defined as a ratio of the processing power that can be used to build a schedule over the globally available processing power. This definition was motivated by the heterogeneity of multi-cluster platforms. In such configurations, expressing a resource constraint as a number of processors that cannot be exceeded during the execution of the schedule is not really relevant as scheduling a PTG on 100 processors computing at 1 GFlop/sec is not the same as on 100 processors computing at 4 GFlop/sec. The allocation procedure then has to dispatch the allowed resource amount among the different tasks of a PTG while ensuring the respect of the resource constraint and minimizing the makespan of the PTG.

Two different procedures, called SCRAP and SCRAP-MAX, were presented in [10]. Both of them starts from an initial allocation of one processor per task. Each iteration of these procedures adds one more processor to the task belonging to the critical path of the PTG that benefits the most from this 1-processor allocation increase. The two procedures differ in the way they detect a violation of the resource constraint. In SCRAP, a violation is detected if the sum of the areas of the tasks, *i.e.,* the product of their execution times by the processing power they use, using the current allocation divided by the time spent executing the critical path of the PTG exceeds β. In other words, this allocation will lead to a schedule that *globally* uses more resources than allowed. In SCRAP-MAX, the application of the resource constraint takes the precedence levels of the PTG into account. The precedence level (l) of a task t is a ($a \geq 0$) if all its predecessors in the PTG are at $l < a$ and at least one of its predecessor is at $l = a - 1$. The idea is to restrain the amount of resources allocated at any precedence level to β. The rationale behind this variant is that, in the mapping step, the ready tasks candidate to a concurrent mapping often belong to the same precedence level. If all these tasks can be executed concurrently, our constraint ensures that the *maximum* processing power usage in that level is less than a β part of the globally available power.

Both allocation procedures were evaluated by simulation over a broad range of application and platform combinations. The resource constraint applied on the scheduling of a single PTG was respected in 99% of the scenarios. This indicates that these procedures allow us to fairly schedule several PTGs concurrently. If an equal share of the available resources is usable for each PTG

schedule, we can indeed guarantee that no particular schedule will consume more resources than allowed and consequently trouble the other applications.

Further experimentations showed that if both allocation procedures respect their initial resource constraints, SCRAP-MAX produces shorter schedules when the constraint is loose (*i.e.,* having a value close to one). Recall that SCRAP builds allocations that globally respect the constraint. A problem can occur when SCRAP determines small allocations for most of the tasks and a few large allocations. This may lead to the post-postponing of those ready tasks allocated on many processors as all the resources they need are not available. As SCRAP-MAX applies the resource constraint on a per level basis, this guarantees that none of the concurrent tasks belonging to a same level will be postponed due to resource unavailability. For this reason we will only consider the SCRAP-MAX allocation procedure in the remaining of this study.

Two questions still remain. The first is to determine in which order the tasks of the different allocated PTGs should be considered for mapping to not compromise the potential fairness allowed by the constrained resource allocations. The second question is to define what value should take the resource constraint of each concurrent PTG to ensure a good fairness and a small makespan. In the next sections we discuss different options for mapping and investigate several strategies to compute β depending on either the number of concurrent PTGs or the characteristics of the different application graphs to find a good balance between both objectives.

5 Concurrent Mapping of Allocated PTGs

To schedule a single PTG, either on homogeneous or heterogeneous platforms, most of the two-step heuristics rely on a list scheduling algorithm during the mapping step. At the end of the allocation step, an ordered list of the tasks is built according to a particular priority criterion. A commonly used ordering is to rank the tasks according to their *bottom level* [14, 15], *i.e.,* the distance to the exit node of the PTG in terms of execution times. In the case of a single PTG, this order guarantees the respect of the precedence relations and favor the task that is the farthest from the end of the application when several tasks are ready at the same time.

When scheduling several PTGs concurrently, the prioritization of tasks is more complex as an aggregation of the different applications into a single PTG is required. This issue has been discussed in [15] in which four aggregation methods into a single DAG have been proposed. Such a global ordering of tasks coming from different applications may have a strong impact on the fairness of the produced schedule. For instance the entry tasks of a small PTG will have a low bottom level and be close to the end of the ordered scheduling list. This PTG will consequently experience a high delay as its entry tasks are ready as soon as it is submitted. Several strategies can be envisaged to prevent such a postponing issue. A classical approach, used by batch schedulers, is to use conservative backfilling strategies [7] that try to fit some waiting tasks into schedule holes to improve resource usage without delaying already mapped tasks. This method that is already complex in the case of independent tasks is even harder to implement in presence of dependencies between tasks. Indeed

the scheduler will not only to find a hole in the schedule in which a task fits but a hole that also respects the precedence constraints.

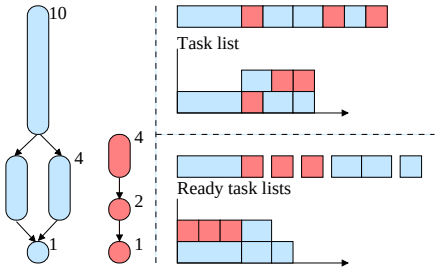

Figure 1: Illustration of the impact of an ordering limited to the list of ready tasks (bottom right) on the schedule length with regard to a global ordering (top rigth).

In this work, we propose to rely on a simpler mapping procedure to prevent the postponing issue mentioned in [15]. This approach still orders tasks according to their bottom level, but only those that are ready. A task is ready only when all its predecessors have finished their executions. Let consider two PTGs, as shown in the left part of Figure 1, one requiring less work than the other and more precisely whose tasks can all be executed during the execution of the first of the other PTG. Let also assume that each PTG is allowed to use a half of the available power, *i.e.,* one processor in that simple example and that there is no backfilling available. The top right part of Figure 1 shows the schedule resulting from a global ordering while the bottom right part presents the schedule obtained by ordering only the ready tasks. We can see that with the global ordering the beginning of the execution of the small PTG is postponed until the completion of the first task of the big PTG. The resulting schedules is thus unfair, as the small application has to wait and also inefficient as it contains idle times. Conversely with the ordering of the ready tasks only, the first task of the small PTG can start immediately to achieve a fairer and more efficient schedule.

The advantage outlined by this simple example is not enough to ensure that postponing will not occur. As the mapping decisions for the smallest PTG, *i.e.,* the one whose entry task has the smallest bottom level, will be taken once all the entry tasks of the other PTGs have been mapped, it could happen that not enough resources are still available leading to the postponing of that task. Thanks to the respect of the resource constraints by our allocation procedure, we expect that this small PTG will still have its share of the resources available when its entry task(s) will be considered for mapping.

Finally, our mapping procedure will select the first task of the list, *i.e.,* the one with the highest priority, and determines the processor set that achieves the earliest finish time. It may happen that a task is delayed because its computed processor allocation is (perhaps only slightly) larger than the number of processors available when the task is actually ready for execution. In practice, this phenomenon introduces idle times in the produced schedules. To prevent the apparition of such holes, we include an *allocation packing* mechanism in our mapping procedure. If a task has to be delayed because all the processors it

needs are not available, we reduce its allocation if and only if the task can start earlier and finish no later than on its original allocation.

6 Determining the Resource Constraint

The first proposed strategy will be used as a baseline competitor in our experiments. It consists to allow each of the submitted PTG to use all the available resources. In other words, this strategy allows each application to have a selfish behavior and let them compete for the resources. It corresponds to fix β to 1 for each application. The motivation of defining such a strategy is to have an indication on the fairness of the schedules built by two-step heuristics such as those previously proposed in the literature [9, 11]. As said in Section 3 these heuristics have been designed to schedule a single application that can potentially use all the available resources. They consequently do not aim at being fair when applied to multiple concurrent applications. We will denote this first selfish strategy by S in the evaluation presented in Section 7.

By opposition the second strategy relies on a simple assumption about fairly sharing resources among concurrent applications. The fairest repartition of the resources, which does not systematically imply the fairest schedule, is to allow each of the submitted applications to use an equal share of the resources to build its own schedule. For instance if ten PTGs have to be scheduled simultaneously, each of them will be associated to a resource constrain β equal to 0.1 and will be thus allowed to use only one tenth of the processing power of the platform. More generally if $\mathcal{A}$ is the set of applications to schedule, each PTG in $\mathcal{A}$ will have to respect a resource constraint $\beta = 1/|\mathcal{A}|$, where $|\mathcal{A}|$ denotes the cardinality of set $\mathcal{A}$. We denote by ES this strategy relying on an equal sharing of the resources.

Allowing each application to use an equal share of the available processing power to build its own schedule may increase the fairness but can also lead to inefficient schedules in terms of makespan is some of the PTGs cannot fully exploit the allocated resources while some others are limited by a too constrained allocation. A solution is to unbalance the sharing of resources so that each PTG is constrained proportionally to its contribution to the set of applications for a particular metric. We propose to study the impact of three different metrics inherent to the structure of a PTG on the fairness and global makespan of the resulting schedule.

The first considered characteristic is the length of the critical path of each PTG. Indeed if an application has a long critical path it could be interesting to allow it to use more resources to reduce the execution time of the different tasks composing the critical path. Conversely, a PTG with a short critical path may not complete earlier with more allocated resources. It has to be recalled that the allocation procedure attributes processors preferentially to tasks belonging to the critical path and thus that using more resources will tend to reduce the critical path length.

The second studied characteristic is the maximal width of each PTG, *i.e.,* the size of the precedence level comprising the most tasks. A large PTG, or at least with one large level, can exploit more task parallelism than a chain-like PTG. If the allocation of a large PTG is too constrained, this large level becomes a bottleneck for the application whether because some tasks have to be postponed as the needed resources are already used by other tasks of the same

level or because the allocations of the different tasks composing this large level have to be reduced to fully exploit the task concurrency. This second situation is likely to occur as our allocation procedure applies the resource constraint on a per-level basis. The sum of the processing power of the determined allocations for the tasks of the largest level of a PTG will thus not exceed a β part of the globally available processing power.

Finally we also consider the respective amount of work of each application to schedule, *i.e.,* the sum of the floating point operations of the tasks composing the PTG. If one of the concurrent PTG has only a little amount of work to do, the heuristic may schedule it on less resources than what it is allowed to use. But the unused resources cannot benefit to other PTGs that require more than their share as the constraint prevent them to use more resources. This will thus limit the amount of data-parallelism that can be exploited by our scheduling algorithm.

To determine which of these three characteristics of the PTG is the most likely to produce fair schedules, we propose a third strategy for the computation of the resource constraint based on the relative contribution of the i^{th} PTG , denoted by γ_i, with regard to the complete set of applications to schedule. In this strategy, called PS (for Proportional Share), the resource constraint associated to each PTG is given by Equation 1.

$$\beta_i = \frac{\gamma_i}{\sum_{j \in \mathcal{A}} \gamma_j}. \tag{1}$$

In the next section, we will distinguish three declinations of the PS strategy corresponding to the investigated characteristics: $PS-cp$ for constraint related to the critical path length, $PS-width$ for that related to the maximal width of a PTG and $PS-work$ when β is proportional to the amount of work performed by each application.

If the PS strategy has been proposed to avoid the wasting of resources that can occur in the ES strategy, it can also lead to unwanted situations that have a negative impact on fairness. For instance if the work of one PTG is very small compared to the total work to schedule, the PS strategy will allow this PTG to use only a few resources to build its schedule. Consequently, its makespan will be much longer than the makespan this PTG could have expected with a dedicated access to the whole platform. To ensure that each PTG can use a reasonable share of the available resources, we propose a fourth strategy to find a compromise between the ES and PS strategies. Independently of the considered characteristic of the PTG, we introduce a tunable parameter μ in the determination of the β constraint to ensure that each PTG can use enough resources to have a good makespan while preventing the wasting of resources by PTGs having a small contribution. The objective is to find a better tradeoff between our two performance criteria that are fairness and makespan reduction. We denote this strategy by WPS (for Weighted Proportional Share). The computation of the β resource constraint for each PTG is given by Equation 2.

$$\beta_i = \frac{\mu}{|\mathcal{A}|} + \frac{(1-\mu)\gamma_i}{\sum_{j \in \mathcal{A}} \gamma_j} \tag{2}$$

Note that the μ parameter takes its values between 0 and 1 and that when $\mu = 0$ (resp. 1) , the resource constraint will be the same as in the PS (resp.

ES) strategy. As for the *PS* strategy, we will distinguish in the next section the $WPS - cp$, $WPS - width$ and $WPS - work$ variants.

7 Evaluation

We resort to simulation to evaluate our proposed algorithms. Simulation allows us to perform a statistically significant number of experiments for a wide range of application an platform configurations (in a reasonable amount of time). We use the SimGrid toolkit [2] 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. Our simulations target the Grid'5000 subsets as described in Section 2. They account for time taken by computation and data redistribution operations.

We rely on two complementary metrics to evaluate the different definitions of the resource constraint proposed in Section 6. We first estimate how fair are the produced schedules. In [15], the fairness is defined on the basis of the slowdown each PTG would experience from resource sharing. The slowdown of an application a is defined as the ratio between the makespan achieved when a has the resources on its own ($M_{own}(a)$) and the makespan of the same application achieved in presence of concurrency ($M_{multi}(a)$). The slowdown value of the application a is then given by

$$Slowdown(a) = M_{own}(a)/M_{multi}(a). \tag{3}$$

A schedule will be considered as fair if each application experiences a similar slowdown. The *unfairness* of a schedule S is thus defined as the sum of the absolute values of the difference between the slowdown of each PTG and the average slowdown. For a set $\mathcal{A}$ of applications to schedule, the average slowdown is given by

$$AvgSlowdown = \frac{1}{|\mathcal{A}|} \sum_{\forall a \in \mathcal{A}} Slowdown(a), \tag{4}$$

while the unfairness is defined as

$$Unfairness(S) = \sum_{\forall a \in \mathcal{A}} |Slowdown(a) - AvgSlowdown|. \tag{5}$$

A low value for unfairness means that each PTG experiences almost the same slowdown, *i.e.,* the schedule is reasonably fair.

The second metric used in our evaluation is the average makespan as a schedule that multiplies by 5 the makespan of each of the submitted applications may be fair, according to the above definition, but also very inefficient. As a simple average over a large range of experiments can smooth results and thus hide some extreme values, we consider the average relative makespan instead. For each experiment, *i.e.,* a platform and a set of PTGs, the makespan achieved by each strategy of determination of the resource constraint is divided by the best makespan achieved for this experiment.

We use the three different types of PTGs presented in Section 2. We recall that the randomly generated PTGs comprise 10, 20 or 50 tasks. The FFT PTGs have 4, 8 or 16 levels (that is 15, 37 and 95 tasks) while all the Strassen PTGs

have the same number of tasks (25). We generate 25 random combinations for each number of concurrent PTGs (2, 4, 6, 8 and 10). As we target four different platforms, we thus have 100 different runs for each scenario. The results presented in the remaining of this section have been obtained by taking the average over these 100 runs.

Before comparing the four strategies of determination of the resource constraint with regard to the fairness and makespan of the produced schedule, we first have to find an adequate value for the μ parameter used in the WPS strategy.

Figure 2 shows the evolution of the unfairness (left) and the average makespan (right) when the μ parameter of the $WPS - work$ strategy varies from 0 to 1 for random PTGs. Note that we do not use the average relative makespan in that figure but a simple average over the 100 runs as only one scheduling heuristic is studied.

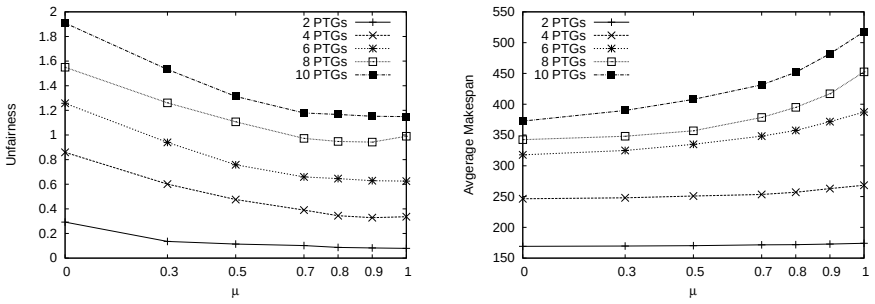

Figure 2: Evolution of the unfairness (left) and the average makespan (right) when the μ parameter of the $WPS-work$ strategy varies from 0 to 1 for random PTGs.

First we can see that the unfairness increases with the number of concurrent PTGs, which is obvious as this metric is the sum of the respective slowdowns of the applications. Then we can see that the two graphs follow opposite trends. The unfairness decreases as μ takes bigger values while the average makespan increases. This indicates that a low value of μ that fixes a different constraint for each PTG proportionally to their respective work, tends to favor the reduction of the total makespan. Conversely a high value of μ that implies a fair repartition of the resources among the PTGs and thus minimizes the unfairness. Finally we can see that for $\mu \geq 0.7$ there is only a little gain in terms of unfairness reduction while the average makespan increases more quickly. Consequently we will fix the value of μ to 0.7 for the $WPS-work$ strategy in the experiments of the remaining of this section in order to have a good balance between makespan reduction and fairness.

We ran the same kind of experiments for the other variants of the WPS strategy and for the other categories of PTGs. For the $WPS - work$ variant, fixing μ to 0.7 is an appropriate value for all kinds of PTG. Similarly, for the $WPS - cp$ variant, we use the same value of μ for each category which is in this case set to 0.5. Finally for the $WPS - width$ variant, the μ parameter takes different values, namely 0.3 for FFT applications and 0.5 for randomly generated PTGs. This difference is due to the structure of the FFT task graphs

that are very regular and whose depth is related to the width. Such a small value of μ will determine the value of β by giving more importance to that width. For the random PTGs, as we have as much "chain" graphs as "fork-join" graphs in our test set, that 0.5 value is quite logical. It has to be noticed that as all the Strassen PTGs have exactly the same shape and thus the same maximal width, the PS and the WPS have absolutely no interest. They indeed lead to exactly the same schedules as those produced by the ES strategy.

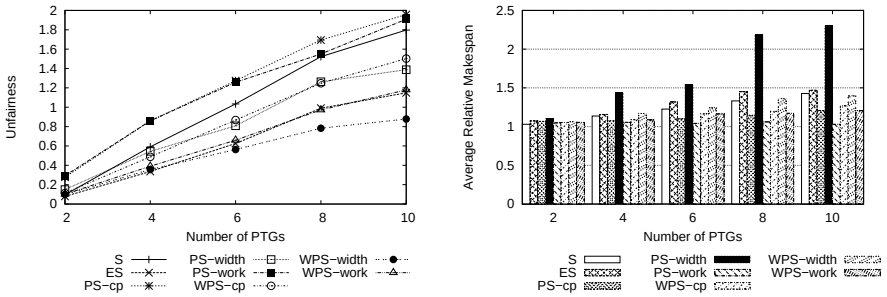

Figure 3: Unfairness (left) and average makespan (right) values (over 100 runs with 2-10 randomly generated PTGs on 4 multi-cluster platforms) of 8 resource constraint determination strategies.

We now can compare the eight resource constraint determination strategies with regard to the unfairness and makespan metrics. Figure 3 shows the results obtained for the randomly generated PTGs. The left part of this figure shows the unfairness while the right part presents the average relative makespan. We recall that the performance of the S strategy, in which the scheduler produces selfish allocations and the PTGs compete for resources, constitutes a baseline reference for the other strategies as it gives an indication of the performance of heuristics originally designed to schedule a single PTG. We can see that five strategies are fairer than the selfish S strategy. These are the ES, *i.e.,* allow the use of an equal share of the available resources to each PTG, the three variants of the WPS strategy, that balance the constraint between an equal share and a distribution related to the PTG respective contribution according to a given characteristic (critical path length, maximal exploitable task parallelism or work) and the $PS-width$ strategy that determines the resource constraint applied to each PTG proportionally to its maximal width. The $WPS-cp$ and $PS-width$ strategies improve the fairness with regard to the selfish competitor of about 16 %, while the ES and $WPS-work$ raise this gain up to 36 %. Finally the $WPS-width$ strategy is the fairest one and increases the fairness by a factor of 2 with regard to a schedule relying on selfish allocations. Furthermore this improvement remains consistent when the number of concurrent PTGs increases from 4 to 10.

It is also interesting to notice that the $PS-cp$ and $PS-work$ strategies, *i.e.,* determine the constraint proportionally to the length of the critical path or to the work a PTG has to accomplish, are less fair than the selfish strategy. In other words, these strategies that aim at limiting the natural competition between the applications to access the resources lead to more harm than benefit. Nevertheless such a apparently negative result can be easily explained through

the following simple example. Consider 10 PTGs scheduled concurrently. Eight of them do not suffer of competition at all while 2 PTGs are delayed and spend 5 times as much time to complete. The slowdown (M_{own}/M_{multi}) of the 8 first PTGs is thus equal to 1 while that of the delayed PTGs is 0.2. Consequently the average slowdown is $(8 \times 1 + 2 * 0.2)/10 = 0.84$ and the unfairness is $8 \times (1 - 0.84) + 2 * (0.84 - 0.1) = 2.56$ which is an high value. The approach followed by the $PS - cp$ and $PS - work$ strategies can lead to such situations as the PTGs are allowed to use resources according to their needs. It means that small applications may have only a few processors to build their schedules if the other applications have longer critical paths or more work to execute. Consequently the task parallelism of these small PTGs may not be exploited leading to longer schedules.

On the right part of Figure 3 we can see that if the $PS - cp$ and $PS - work$ strategies are the least fair ones, they are also the ones achieving the best makespans. The explanation of this result is that if the small PTGs subjected to delays due to tight resource constraints have a strong impact on fairness, they only have a little influence on the completion time of a particular run. Indeed if these PTGs are only allowed to use a few resources it is because their critical path is short or the work they have to execute is small, and thus their total execution time is quite insignificant with regard to their competitors. Then we observe that the relative makespan achieved by the S strategy increases with the number of concurrent PTGs. In other words, the selfishly determined allocations become a problem when the competition for resources gets harder. As not enough resources are available to execute all the ready tasks concurrently, some of them are postponed and the makespans of some PTGs then increase.

This second metric relative to makespan also allows us to see if the fairness achieved by a given strategy does not come at a too high price in terms of global completion time. For instance, if the $PS - width$ and $WPS - cp$ strategies lead to similar fairness, the average relative makespan of the $PS - width$ constraint determination method is almost twice as big as that of $WPS - cp$. We can also find a winner between the ES and $WPS - work$ strategies. As mentioned in Section 6, the ES strategy ensures fairness by allowing each PTG to build its schedule on an equal share of the available power but at the price of the wasting of some resources. The consequence of this choice is that the respective lengths of the schedule of the different PTGs are longer even if the overhead is similar for all of them. Conversely the $WPS - work$ strategy distributes the resources in a way that achieves a similar fairness but also good makespans. In comparison to the average relative makespans achieved by the S strategy, those of the $WPS - work$ are between 4% and 15% closer to the best solution produced while the relative makespans achieved by the ES are higher. Finally, the best strategy in terms of fairness, namely the $WPS - width$ strategy, offers to increase the fairness by a factor of 2 while achieving makespans competitive to those of the selfish strategy. When compared to the solution leading to the smallest makespans, the schedules produced by $WPS - width$ are on average 16 % longer but 55 % fairer.

Figure 4 shows similar results for FFT PTGs. The main characteristic of these application graphs is their regularity as every tasks in a given level have the same cost. Furthermore the different PTGs have the same structure and only differ by the number of tasks they comprise. Consequently there as less dissimilarities between the concurrent applications than for random PTGs. This

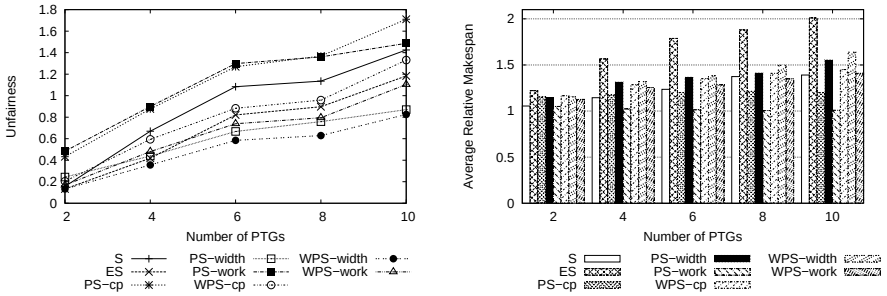

Figure 4: Unfairness (left) and average makespan (right) values (over 100 runs with 2-10 FFT PTGs on 4 multi-cluster platforms) of 8 resource constraint determination strategies.

has an impact on the unfairness achieved by the different strategies which is lower than for random PTGs. We can see the same trends and almost the same ranking of the strategies with regard to the fairness and relative makespan metrics as in Figure 3. The main difference is that the $PS-width$ is now the second best heuristic in terms of fairness and even leads to shorter schedules than its weighted variant while it produced the worst schedules in terms of makespan for randomly generated PTGs. We can also see that the S strategy is more competitive for this class of applications especially in terms of makespan. In opposition to the random PTGs, FFT graphs have only a limited amount of task parallelism to exploit that depends on the number of levels (4, 8 or 16). Consequently the scheduler is less often faced to situations where many ready tasks have to be mapped concurrently. This implies a less severe competition for resources between applications and thus a better fairness conjugated to better makespans. This is especially true when the constraint can be adapted to the amount of task parallelism that can be exploited as in the $PS-width$ and $WPS-width$ strategies. We can also observe that the ES strategy achieves particularly poor performance in terms of makespans for these PTGs (schedules up to twice as long on average as the best solution for 10 concurrent PTGs). Finally the $WPS-work$ strategy offers a good balance between fairness (third best) and schedule length (competitive with the S strategy) up to 8 concurrent PTGs. For 10 concurrent submissions this strategy becomes less fair and the $PS-width$ should be preferred.

Figure 5 presents the unfairness (left) and average relative makespan (right) achieved by the different strategies on Strassen PTGs. As FFT graphs, the Strassen PTGs are very regular but they also have a fixed structure implying the same number of tasks, and the same maximal width. Then all the concurrent PTGs only differ in the costs of the different tasks composing them. Consequently the strategies based on the respective widths of the PTGs have no interest for such applications. This additional feature nevertheless improves the respective fairness of the other studied strategies. We can see that for this particular application, the $WPS-work$ strategy is not as close to the ES strategy as for the previous ones, but less fair of around 25 %. Nevertheless the gain on the makespan of $WPS-work$ over ES is still important (around 35 %).

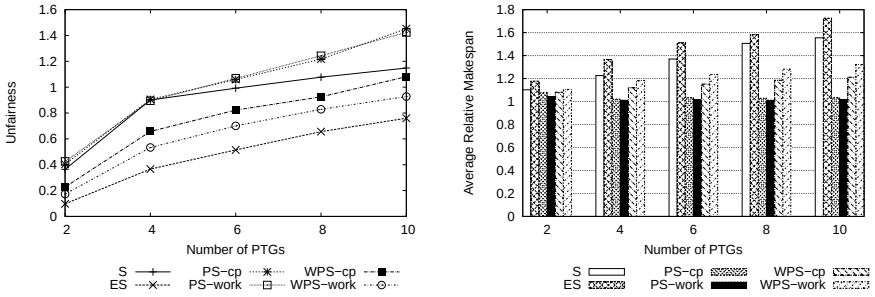

Figure 5: Unfairness (left) and average makespan (right) values (over 100 runs with 2-10 Strassen PTGs on 4 multi-cluster platforms) of 8 resource constraint determination strategies.

The conclusions to this evaluation are that the proposed $WPS - width$ strategy to determine a constraint limiting the amount of processing power each concurrent PTG can be allocated on, gives better results in terms of fairness than a selfish allocation strategy. It also leads to makespans comparable to (or in some cases better than) those of the selfish approach. We also found that the $PS - work$ strategy is the least fair one but nevertheless the heuristic producing the shortest schedules. Even if such a result was not our primary objective (which was to design a fair multiple PTG scheduling heuristic), it however has to be considered as this unfair strategy can allow the scheduler to increase the throughput of the platform it manages.

8 Conclusion and Future Work

Heuristics for scheduling applications structured as parallel task graphs (PTGs) on a heterogeneous parallel computing platform such as a multi-cluster have been developed in the case of a dedicated platform [1, 9, 11]. However, multi-cluster platforms are commonly shared by multiple users submitting their applications that then have to compete for resources. In this context, to the best of our knowledge, the only previously proposed work is a study on the design of a constrained resource allocation procedure for two-step scheduling algorithms [10]. The idea of that work was to limit the amount of resources an application can use to build its schedule in order to leave enough room for the execution of its competitors. In this study we aimed at completing this study by proposition a mapping procedure in which only the ready tasks are sorted according to their distance to the end of the application to prevent the postponing of tasks belonging to small PTGs.

We also investigated eight strategies for the determination of the resource constraint. Two of them relies on basic sharing ideas: be selfish or share evenly. Then we proposed strategies that share the resources between the PTGs proportionally to their contribution on one of the three following characteristics: critical path length, maximal task parallelism to exploit or amount of work to execute. For each of these characteristics, the determination of the resource constraint can be balanced by taking the number of concurrent applications into account.

Finally we evaluated these different strategies through simulation over a large range of scenarios. Our main finding is that a constraint that is a good balance between attributing similar shares of resources to the concurrent PTGs and taking into account the relative importance of the maximal amount of task parallelism that can be exploited by each application allows to build schedules that are fair without paying the price of longer completion time. This is the approach followed by the proposed $WPS - width$ strategy. In the particular case in which all the concurrent PTGs have the same shape, *e.g.,* for the Strassen matrix multiplication algorithm, taking their relative amount of work into account, as in the $WPS - work$ strategy leads to the best tradeoff. We also shown that a strategy strictly based on the relative importance of the work, as the $PS - work$ strategy, leads to unfair but very short schedules that can present a certain interest for resource managers.

As a future work, we plan to consider different submission times for the concurrent PTGs. Such a configuration is more representative of what happens on real-world platforms but it is also very challenging. This indeed implies that the resource constraints have to be modified on the arrival of a new application in the system and thus that the schedules of the already running applications may have to be reconsidered. Being able to dynamically recompute the respective resource constraints and modify the schedules accordingly should also allow us to react to the completion of a given application in order to redistribute its share of resources to the other PTGs.

References

[1] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *10th International Euro-Par Conference*, volume 3149 of *LNCS*, pages 230–237. Springer-Verlag, August 2004.

[2] 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, March 2008.

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

[4] H. Chen and M. Maheswaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Systems. In *12th Heterogeneous Computing Workshop (HCW)*, Fort Lauderdale, FL, 2002.

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

[6] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling on Heterogeneous Core Systems. In *First Workshop on Operating System Support for Heterogeneous Multicore Architectures*, Brasov, Romania, 2007.

[7] D Feitelson, L Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In *Job Scheduling Strategies*

for Parallel Processing Workshop (JSSSP), volume 1291 of *LNCS*, pages 1–34, Geneva, Switerland, April 1997. Springer.

[8] M. A. Iverson and F. Özgüner. Hierarchical, Competitive Scheduling of Multiple DAGs in a Dynamic Heterogeneous Environment. *Distributed System Engeneering*, (3), 1999.

[9] T. N'Takpé and F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In *12th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 3–10, Minneapolis, MN, July 2006.

[10] T. N'takpé and F. Suter. Self-Constrained Resource Allocation for Parallel Task Graph Scheduling on Shared Computing Grids. In *19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Cambridge, MA, November 2007.

[11] T. N'takpé, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *6th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 250–257, Hagenberg, Austria, July 2007.

[12] A. Radulescu and A. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *15th Int. Conf. on Parallel Processing (ICPP)*, Valencia, Spain, 2001.

[13] F. Suter. DAG Generation Program. `http://www.loria.fr/~suter/dags.html`.

[14] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE TPDS*, 13(3):260–274, 2002.

[15] H. Zhao and R. Sakellariou. Scheduling Multiple DAGs onto Heterogeneous Systems. In *15th Heterogeneous Computing Workshop (HCW)*, Island of Rhodes, Greece, April 2006.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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