Self-Constrained Resource Allocation Procedures for Parallel Task Graph Scheduling on Shared Computing Grids

Tchimou N'Takpé, Frédéric Suter

SELF-CONSTRAINED RESOURCE ALLOCATION PROCEDURES FOR PARALLEL TASK GRAPH SCHEDULING ON SHARED COMPUTING GRIDS

Tchimou N'Takpé and Frédéric Suter
Nancy Université / LORIA
UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1
Campus scientifique - BP 239, F-54506 Vandoeuvre-lès-Nancy
email: {Tchimou.Ntakpe,Frederic.Suter}@loria.fr

ABSTRACT
Two of the main characteristics of computation grids are their heterogeneity and the sharing of resources between different users. This is the cost of the tremendous computing power offered by such platforms. Scheduling several applications concurrently in such an environment is thus challenging. In this paper we propose a first step towards the scheduling of multiple parallel task graphs (PTG), a class of applications that can benefit of large and powerful platforms, by focusing on the allocation process. We consider the application of a resource constraint on the schedule and determine the number of processors allocated to the different tasks of a PTG while respecting that constraint. We present two different allocation procedures and validate them in simulation over a wide range of scenarios with regard to their respect of the resource constraint and their impact on the completion time of the scheduled applications. We find that our procedures provide a guarantee on the resource usage for a low cost in terms of execution time.

KEY WORDS
Scheduling, Grid computing, PTG, Allocation.

1 Introduction

Currently deployed grid computing platforms hold the promise of higher levels of scale and performance than possible with a single cluster due to improvements in network and middleware technology. The costs of this tremendous computing power right behind the plug are the heterogeneity of the resources that compose such platforms and the competition to access the resources between multiple users. Such a context raises the question: how to concurrently schedule the applications of several users while minimizing the perturbations between applications and getting the best "bang for the buck" 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 [11], or hierarchical schedulers in which applications are first dispatched among clusters and then relying on waiting queues algorithms [4, 5]. A limitation of these scheduling algorithms is that they assume that the applications graphs only comprise sequential tasks. But a way to take a higher benefit from the large computing power offered by grids 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) which 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 platforms exist [1, 2, 7] but they consider that all the platform is available when building the schedule of a single application. Consequently these heuristics may produce schedules that require a lot of resources. In a shared environment such schedules can negatively impact (or be impacted by) other scheduled applications. The most successful approaches proceed in two phases: one phase to determine how many processors should be allocated to each data-parallel task, another phase to place these tasks on the platform using standard list scheduling algorithms.

In this paper, we make a first step in the design of scheduling heuristics of PTGs on a shared platform by focusing on the allocation procedure. A constraint on the resource amount that can be used to schedule a given application can be fixed either by the application provider or a meta-scheduler responsible of the scheduling of the multiple applications that share the platform. We propose two allocation procedures aiming at respecting such resource constraints expressed as a ratio of the processing power that can be used to build the schedule over the globally available processing power of the platform. The first procedure, called SCRAP, ensures that the total amount of resources allocated to the tasks of the PTG respects the constraint while in the second procedure, called SCRAP-MAX, we guarantee that the maximum processing power that can be used at any precedence level of the PTG does not exceed what is allowed by the resource constraint.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents our allocation procedures, which we evaluate in Section 4. Section 5 concludes the paper with a summary of our findings and perspectives on future directions.

2 Related Work

Several authors have studied the concurrent scheduling of multiple applications onto heterogeneous platforms [4, 5, 11]. Authors of [11] 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. This work requires all applications to be submitted to a single scheduler at the same time, while we follow a multiple independent schedulers approach leading to a potentially dynamic submission of PTGs. A two-level distributed scheduling algorithm for multiple DAG 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 paper is more on environment-related issues, *e.g.*, task arrival and machine failure rates or wait queue sizes, than on scheduling concerns, like ensuring a fair access to the resources for instance. The hierarchical competitive scheduling heuristic for multiple DAGs onto heterogeneous platforms provided in [5] is the most related work as it also 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 the core of the present work, does not arise in these researches.

On the other hand, two heuristics were recently proposed: HPCA (Heterogeneous CPA) [7] and MHEFT (Mixed-parallel HEFT) [2] to schedule a single PTG on a heterogeneous platform. HCPA extends the CPA algorithm [9] 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 [10] to the case of data-parallel tasks. MHEFT performs list-scheduling by reasoning on average data-parallel task execution times for 1-processor allocations on all possible clusters. Weaknesses in both HCPA and MHEFT were identified and remedied in [8], 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 lead to schedules that would be preferred by the majority of users as it achieves a cost-effective trade-off between application makespan and parallel efficiency (*i.e.*, how well resources are utilized). In this work we compare our approach with the HCPA algorithm and to one of the MHEFT variants proposed in [8], called MHEFT-MAX, in which no task allocation on a cluster can be larger than some fraction of the total number of processors in that cluster.

3 Self-Constrained Resource Allocation

In this section we focus on the first phase of a two step scheduling algorithm in which the number of processors allocated to each of a PTG is determined. In the second step, a classical list scheduling algorithm is used to place each allocated task on a specific processor set.

The determination of a resource constraint applied on the scheduling of a given application can be done either by the provider of each application or by a central meta-scheduler. Leaving the responsibility of the constraint determination to users may lead to selfish behaviors, *i.e.*, a loose constraint for each application, that can compromise an efficient concurrent execution of the different applications. Conversely a meta-scheduler will be responsible to adapt the resource constraint on each independent schedule depending on the global load of the environment.

A constraint on the resources that can be used to schedule an application can be expressed in several ways mainly depending on the platform model. In this paper, we consider a computing platform that consists of c clusters, where cluster $C_k, k = 1, \ldots, c$ contains p_k identical processors. A processor in cluster C_k computes at speed r_k, which is defined as the ratio between that processor's computing speed (in operations per seconds) to that of the slowest processor over all c clusters, which we call the reference processor speed s_{ref}. Clusters may be built with different interconnect technologies and are interconnected together via a high-capacity backbone. Each cluster is connected to the backbone by a single network link. Inter-cluster communications happen concurrently, possibly causing contention on these network links. For more details on this platform model, the reader is referred to [8].

In this model, the resource constraint can be expressed in terms of a number of processors that cannot be exceeded during the execution of the schedule. This number of processors can be either a maximal value, *e.g.*, the schedule never uses more than X processors at the same time, or an average value, *e.g.*, the schedule cannot allocate more than X processors per task on average. But on heterogeneous platforms, reasoning solely in terms of number of processors is not really relevant as scheduling a PTG onto 100 processors computing at 1 GFlop/sec is not the same as onto 100 processors computing at 4 GFlop/sec. The resource constraint can also be expressed in terms of a ratio of the processing power (maximum or average) that can be used to build the schedule over the globally available processing power. As this expression of the resource constraint on the allocation process seems more adapted to heterogeneous platforms, we use it in our allocation procedures and denote it by β ($0 < \beta \leq 1$). β can be interpreted as the utilization of the resources: $\beta = (used\ power)/(total\ power)$.

The question is now to determine how to dispatch this usable processing power between the different tasks of the PTG while respecting the constraint. We propose two different strategies that both rely on the concept of the

reference cluster of HCPA [7]. The reference cluster is a virtual homogeneous cluster with P_{ref} processors, equivalent to the heterogeneous platform. We denote by $p^{ref}(t)$, the current number of processors allocated to a task t in the reference cluster and by $T^{ref}(t, p^{ref}(t))$, the corresponding predicted computation time. Finally the function $f(p^{ref}(t), t, k)$ is used to determine the actual allocation of task t on cluster C_k from its reference allocation.

The main idea of our first procedure, called SCRAP (Self-Constrained Resource Allocation Procedure), is to determine the allocation of each task while ensuring the respect of the usage constraint β, starting from an initial allocation of one processor per task. In each iteration of the procedure we allocate one more processor the task belonging to the critical path that benefits the most of the addition of a processor to its reference allocation. This iterative process will stop if a violation of the resource constraint is detected as follows.

Let $\omega(t)$ be the work of a task t, *i.e.*, the product of its execution time by the processing power it uses, and $\omega^* = \sum_{t \in \mathcal{N}} \omega(t)$ the total work of the application using the current allocation. To obtain a estimation of the resource amount consumed by this allocation, we divide ω^* by the time spent executing the critical path of the PTG – that is a rough estimation of its total execution time, or *makespan* before the placement phase. We thus define β' as the ratio of $\omega^*/makespan$ over the total processing power of the platform. That β' can be seen as a dynamic expression of the resource constraint evolving along with the allocation. If β' exceeds β, this mean that a violation of the initial resource constraint occurred and that the allocation process must be stopped and the last processor addition canceled. If $\beta' > \beta$ from the initial allocation, SCRAP does not allocate more processors to any task of the PTG.

It is also important to note that an allocation on the reference cluster may end up being so large that it cannot be translated into any feasible allocation on any cluster C_k. For this reason, we stop increasing allocations on the reference cluster when no cluster C_k could accommodate a translation of the reference allocation. This additional stop condition is called *saturated critical path*. Our first allocation procedure is summarized in Algorithm 1.

Algorithm 1 SCRAP

1: **for all** $t \in \mathcal{N}$ **do**
2: $\quad p^{ref}(t) \leftarrow 1$
3: **end for**
4: **while** $\beta' < \beta$ and $\neg$ (saturated critical path) **do**
5: $\quad t \leftarrow$ {critical task $|(\exists C_k | \lceil f(p^{ref}(t), t, k) \rceil < P_k)$ and $\left(\frac{T^{ref}(t,p^{ref}(t))}{p^{ref}(t)} - \frac{T^{ref}(t,p^{ref}(t)+1)}{p^{ref}(t)+1}\right)$ is maximum}
6: $\quad p^{ref}(t) \leftarrow p^{ref}(t) + 1$
7: $\quad$ **Update** β'
8: **end while**

In our second allocation procedure, we modify the application of the resource constraint by taking the precedence levels of the PTG into account. The precedence level ($plev$) of a task t is a ($a \geq 0$) if all its predecessors in the PTG are at $plev < a$ and at least one of its predecessor is at $plev = a - 1$. The main idea of this second allocation procedure is to restrain the amount of resources allocated at any precedence level to β. The rationale behind this variant is that, in the placement phase, ready tasks candidate to a concurrent placement 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 $\beta \times P_{ref} \times s_{ref}$.

Algorithm 2 SCRAP-MAX

1: **for all** $t \in \mathcal{N}$ **do**
2: $\quad p^{ref}(t) \leftarrow 1$
3: **end for**
4: **while** $\beta' < \beta$ and $\neg$ (saturated critical path) **do**
5: $\quad t \leftarrow$ {critical task $|$ $(\exists C_k | \lceil f(p^{ref}(t), t, k) \rceil < P_k)$ and $(plev\_Alloc(t) \times s_{ref}) < (\beta \times P_{ref} \times s_{ref})$ and $\left(\frac{T^{ref}(t,p^{ref}(t))}{p^{ref}(t)} - \frac{T^{ref}(t,p^{ref}(t)+1)}{p^{ref}(t)+1}\right)$ is maximum}
6: $\quad p^{ref}(t) \leftarrow p^{ref}(t) + 1$
7: $\quad$ **Update** β'
8: **end while**

This additional constraint impacts the selection of the critical task to which allocate one more processor. To be candidate, a task may show a maximal benefit of the additional processor as in SCRAP but now the sum of the processing power alloted to the tasks, including itself, in its precedence level ($plev\_Alloc(t) \times s_{ref}$) has also to be less than $\beta \times P_{ref} \times s_{ref}$. Once again the initial allocation can violate the resource constraint. Algorithm 2 shows our second allocation procedure, called SCRAP-MAX.

4 Evaluation

We use simulation to explore wide ranges of application and platform scenarios in a repeatable manner and to conduct statistically significant numbers of experiments. Our simulator is implemented using the SIMGRID toolkit [6].

We consider platforms that consist of 1, 2, 4, and 8 clusters. Each cluster contains a number of processors between 16 and 128, picked at random using a uniform probability distribution. The links connecting the processors of a cluster to that cluster's switch can be Gigabit Ethernet (bw $= 1Gb/s$ and lat. $= 100\mu sec$) or 10Gigabit Ethernet (bw $= 10Gb/s$ and lat. $= 100\mu sec$) and we simulate contention on these links. The switch in a cluster has the same bandwidth and latency characteristics at these network links, but does not experience contention. The links connecting clusters to the network backbone have a bandwidth of $1Gb/s$ and a latency of $100\mu sec$. Half the clusters use Gigabit Ethernet devices, and the other half use 10Gigabit Ethernet devices. Finally, the backbone connecting the clusters together has a bandwidth of $25Gb/s$ and a latency of $50msec$.

In our experiments we choose to keep the network characteristics fixed and we vary processor speeds to ex-

periment with various communication/computation ratios of the platform. Processor speeds, which are measured in GFlop/sec and are homogeneous within each cluster, are sampled from a uniform probability distribution as follows. We consider a fixed number of possible minimum speeds: 0.25, 0.5, 0.75, and 1; and of heterogeneity factors: 1, 2, 5 (when there are more than one cluster in the platform). The maximum processor speed is computed as the product of a minimum speed by a heterogeneous factor. For instance, a minimum speed of 0.5 and a heterogeneity factor of 5 means that the processors have uniformly distributed speeds between 0.5 and 2.5 GFlop/sec. We assume that each processor has a 1GByte memory. The above parameters lead to 40 platform configurations. Since there are random components, we generate five samples for each configuration, for a total of 200 different sample platforms.

We take a simple approach to model data-parallel tasks. We assume that a task operates on a data set of n double precision elements. The volume of data communicated between two tasks is proportional to n. We model the computational complexity of a task as one of the three following forms, which are representative of many common applications: $a \cdot n$ (*e.g.*, image processing of a $\sqrt{n} \times \sqrt{n}$ image), $a \cdot n \log n$ (*e.g.*, , a n element array sort), $n^{3/2}$ (*e.g.*, multiplication of $\sqrt{n} \times \sqrt{n}$ matrices), where a is picked randomly between 2^6 and 2^9. As a result this exhibits different communication/computation ratios. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three. Finally, we assume that a fraction α of a task's sequential execution time is non-parallelizable, with α uniformly picked between 0% and 25%.

We consider applications that consist of 10, 20, or 50 data-parallel tasks. We use four popular parameters to define the shape of the DAG: width, regularity, density, and "jumps". The width determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2, 0.5, and 0.8 for width and 0.2 and 0.8 for regularity and density. Finally, 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). Overall, we have 432 different DAG types. Since some elements are random, for each DAG type we generate three sample DAGs, for a total of $1,296$ DAGs.

We first evaluate the algorithms using our allocation procedures with regard to the respect of the initial resource constraint by comparing the ratio of the processing power actually consumed by a schedule over the globally available processing power with β. If the schedules produced using our allocation procedures (almost) never violate the resource constraint imposed by a user, this work should be conducted one step further to consider the design of a placement procedure able to concurrently place several allocated PTGs, each respecting its resource constraint.

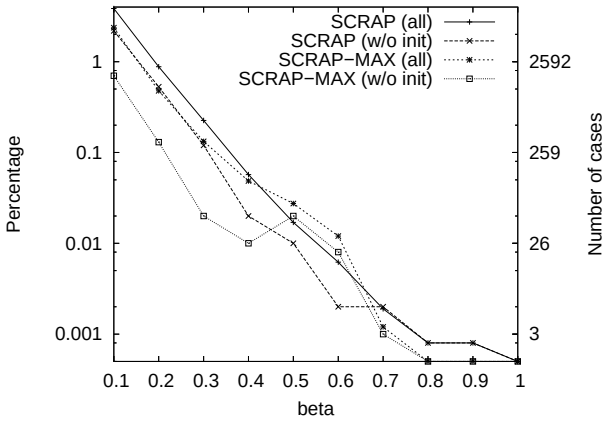

Figure 1. Respect of the β resource constraint by SCRAP and SCRAP-MAX for all simulation runs (all) and when not considering the cases in which the constraint is violated by the initial allocation (w/o init).

Figure 1 presents the percentage of simulation runs that does not respect the initial resource constraint for different values of β ranging from 0.1 to 1. The right vertical axis gives the number of cases corresponding to the percentage indicated on the left vertical axis. For each procedure we distinguish the cases for which the initial allocation violates the constraint (w/o init) as we do nothing in such cases. As expected, the scheduling algorithms relying on our allocation procedures respect the resource constraint in 99% of the cases for $\beta \geq 0.2$. Even when the resource constraint is extreme ($\beta = 0.1$) and considering the initial violations, the constraint is respected in more than 96% of the cases. We can conclude that our allocation procedures guarantee the resource usage of the produced schedules.

To complete this first evaluation, we measured the average and maximum deviation of the resource usage with regard to the constraint for each value of β. When violating the resource constraint, SCRAP (resp. SCRAP-MAX) consumes at most 5.4% (resp. 7.4%) more resources than what is allowed while the average over all β values for both procedures exceeds the initial constraint by less than 3%.

As said before, the MHEFT-MAX scheduling heuristic [8] also proposes to constrain the resources that can be allocated to a task as no allocation on a cluster can be larger than some fraction of the total number of processors in that cluster. This way of limiting the resource usage of the scheduling algorithm is more local and does not offer any guarantee on the processing power used by the complete

schedule mainly because the MHEFT-MAX constraint is expressed in terms of number of processors.

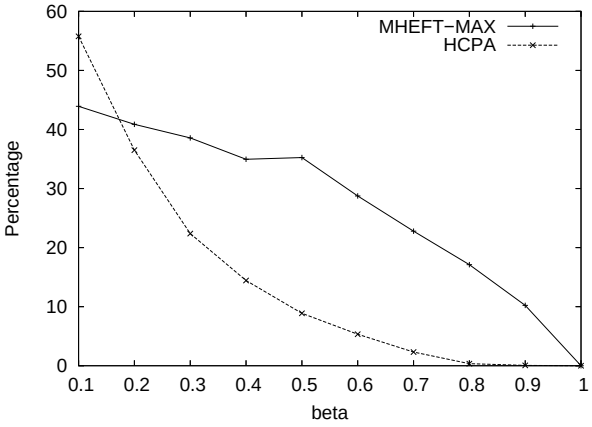

Figure 2. Respect of the β resource constraint by MHEFT-MAX and HCPA.

Figure 2 shows the percentage of simulation runs for which the corresponding global constraint is not respected. For instance, for $\beta = 0.2$, we count the number of runs of MHEFT-MAX-0.2 producing schedules that use more than 20% of the available processing power. We can see on this figure that the percentage of runs that violate the global resource constraint is one order of magnitude higher than for SCRAP and SCRAP-MAX. This confirms our choice of expressing the resource constraint as a power ratio instead of a maximal number of processors.

We also depict in Figure 2 how a *single*-PTG scheduler, such as HCPA, will behave in a shared environment. For each value of β we count the number of cases for which the selfish schedule of HCPA consumes more processing power than what is allowed by the resource constraint. As for MHEFT-MAX, that percentage is very high when the constraint is tight (small values of β) and decreases when we relax the resource usage. One may rightly object that HCPA was not designed to schedule PTG on shared environments, but with this figure we aim at showing the benefit of designing scheduling algorithm dedicated to shared environments instead of using existing algorithms.

We finally evaluate our allocation procedures with regard to the slowdown experienced by a PTG as a result of constraining the resource that can be used to schedule it (as opposed to the makespan achieved when the PTG is having all the resources available). We define the slowdown of an application a as

$$Slowdown(a) = M_{all}(a)/M_{\beta}(a) \quad (1)$$

where M_{all} is the makespan of the PTG when it can use the resources without constraints and M_{β} is the makespan of the same PTG when schedule under a constraint β.

In Figure 3, $M_{all}(a)$ is the makespan achieved by the schedule produced for PTG a under a $\beta = 1$ resource constraint. This figure shows the average slowdown of SCRAP

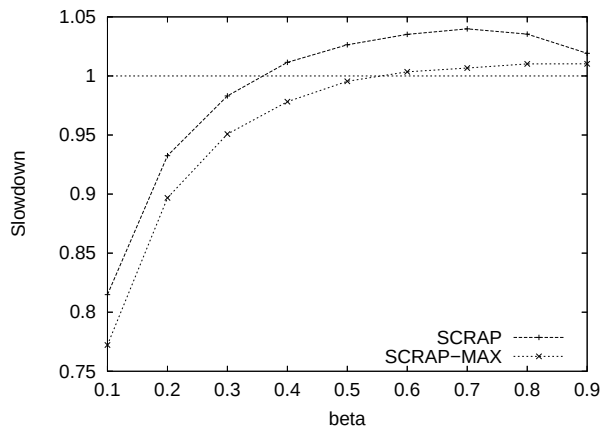

Figure 3. Average slowdown for SCRAP and SCRAP-MAX when β varies. The reference makespan is achieved using $\beta = 1$.

and SCRAP-MAX over the whole range of scenarios when β varies. That slowdown values are very high for both procedures which signifies that scheduling an application while guaranteeing a resource constraint as only a small impact on makespan. For instance, allowing the allocations to use only 10% of the available processing power will only lengthen the schedule by 18% (resp. 23%) for SCRAP (resp. SCRAP-MAX).

It is also very interesting to notice that constraining the allocations may improve the placement step and thus produce shorter schedules, as depicted in Figure 3 by the slowdown values greater than 1. Indeed respecting the resource constraint leads to smaller allocations for some tasks and allow their concurrent execution that is not possible with larger allocations determined without any constraint.

We continue this study of the impact of the resource constraint on performance in Figure 4, in which the makespan achieved by HCPA was used to compute the average slowdown of SCRAP and SCRAP-MAX.

We can see that our allocation procedures lead to shorter schedules than those produced by HCPA for $\beta \geq 0.4$ while for more restrictive constraints, the slowdown is at least of 20% (when $\beta = 0.1$). The better performance achieved by SCRAP and SCRAP-MAX can be easily explained by looking at the energy, *i.e.*, the product between execution time and processing power, consumed by our algorithms and HCPA. When SCRAP, or SCRAP-MAX, achieves a better makespan than HCPA it is only because it uses more processing power to schedule the PTG, while respecting the resource constraint all the same. For instance, when $\beta = 0.8$, SCRAP-MAX consumes 16% more energy than HCPA. This also denotes how conservative the allocation procedure of HCPA is. In Figure 2 we can see that HCPA consumes less than 30% of the available processing power in more than 75% of its schedules. Nevertheless such a conservative behavior does not allow HCPA to guarantee the respect of a resource constraint as SCRAP and

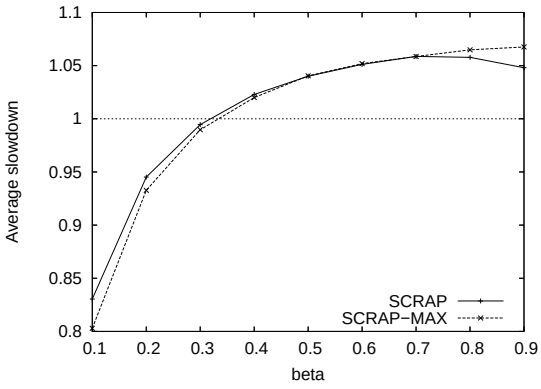

Figure 4. Average slowdown for SCRAP and SCRAP-MAX when β varies. The reference makespan is achieved by the HCPA algorithm.

SCRAP-MAX do.

5 Conclusion and Future Work

In this paper we have studied the scheduling of multiple Parallel Task Graphs (PTGs) onto a shared heterogeneous platform. We made a first step towards this objective by focusing on the allocation procedure in which the number of processors to allocate to each task of the PTG is determined. We handled the concurrent access to resources by several applications by imposing a resource constraint on the schedule. We then proposed two procedures, called SCRAP and SCRAP-MAX, that determine allocations while respecting that constraint expressed as a ratio of the available processing power of the target platform. We validated the schedules deriving our the computed allocations using simulation with regard to the respect of the resource constraint and the impact on the application completion time. First results are convincing are the constraint is respected in 99% of our experiments and with only a small lost of performance. We finally compared our heuristics to the HCPA algorithm [7, 8]. We observed that the allocations of SCRAP and SCRAP-MAX are less conservative than those of HCPA and may lead to shorter schedules while guaranteeing the respect of the resource constraint.

As said before, the present work is a first step towards the scheduling of multiple concurrent parallel task graphs. An interesting future work is to design a placement procedure that takes several allocated PTGs as input, each allocation respecting a resource constraint. Some challenging issues arise in the design of this *multiple*-PTGs scheduler such as ensuring fairness between applications, defining priority functions to favor applications scheduled under tighter constraints, or allowing the dynamic submission of PTGs and thus adapting the resource constraint to the new load conditions. We also plan to compare the schedules by this two-step scheduling algorithm to those obtained using the DAG combination approach of [11], adapted to PTGs.

References

[1] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2003.

[2] 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.

[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. Mahesawaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Systems. In *12th Heterogeneous Computing Workshop (HCW'02)*, Fort Lauderdale, FL, April 2002.

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

[6] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, Tokyo, May 2003.

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

[8] 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)*, Hagenberg, Austria, July 2007.

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

[10] 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.

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