# Fulfilling Task Dependence Gaps for Workflow Scheduling on Grids

Luiz F. Bittencourt
State University of Campinas
Institute of Computing
P.O. 6176, Campinas, São Paulo, Brazil
bit@ic.unicamp.br

Edmundo R. M. Madeira
State University of Campinas
Institute of Computing
P.O. 6176, Campinas, São Paulo, Brazil
edmundo@ic.unicamp.br

## Abstract

*The workflow programming paradigm has had a representative growth in the last years. This model is useful to represent flows of control and facilitate the complexity management of processes that have multiple dependent tasks. With the emergence of e-Science, workflow is becoming a standard for management of scientific processes with massive data sets. Within the workflow execution, scheduling of tasks is primordial to provide efficiency and to speed up the process results arrival. In this paper we consider the execution environment as being a computational grid, which is dynamic, non-dedicated, and has heterogeneous resources. We present a strategy for scheduling dependent task processes, dealing with scheduling and execution of more than one process at the same time potentially using resources in common. The algorithm is dynamic and adaptive, rescheduling tasks that are on the queue of resources not presenting good performance. Simulations show that the proposed strategy can give better schedules by enhancing the resources usage.*

## 1. Introdution

Grids emerged in the middle of 1990's [4] and evolved to a powerful environment for parallel processing and data storage. These characteristics match with e-Science requirements: high processing power and management of large data sets. Also, the workflow paradigm helps in developing distributed systems architectures and algorithms for executing such processes. Thus, the combination of grids and workflows is a promising way of doing science [10].

To improve process execution in grids, whilst heterogeneity means complexity when selecting resources to execute tasks, the dynamic behaviour (variations in resources performance and availability) means complexity in performance prediction and rescheduling. Scheduling, performance prediction and rescheduling are receiving substantial attention from the grid researchers nowadays [2, 7, 9]. However, an important point in scheduling is not receiving the deserved attention: the concurrent execution and scheduling of processes with dependent tasks.

A schedule of a process composed of dependent tasks have gaps between execution of tasks, since tasks must wait for the data from its predecessors to start its execution. When scheduling more than one process in the same set of resources, the algorithm can consider these gaps to execute tasks of processes that arrive. As a consequence, the complexity increases and there are more combinations of possible schedules, whilst the process execution times can be decreased and the overall machine utilization maximized.

In this work we study the aspects cited above to have better schedules for processes on a grid environment. We mix a scheduling heuristic, a dynamic scheduling strategy with performance oriented adaptive task allocation, and we propose an algorithm to use the gaps to execute tasks. Using queue and gaps to execute tasks, the algorithm can estimate the finish time of the process more precisely than when executing processes at the same time on the same resource. With many processes arriving, the first process can delay in an unpredictable way if it shares resources with the arriving processes. This way, one cannot have a good estimation of the execution time of his/her process when it is submitted, what can be achieved using queues.

The paper is organized as follows. In the next section we introduce the task scheduling problem, whilst section 3 shows some related works. An overview of the used base algorithms is shown in Section 4, and the proposed gap searching algorithm is presented in Section 5, in conjunction with a rescheduling strategy. In Section 6 we show some experiments with the proposed algorithm, and Section 7 concludes the paper.

## 2. Task Scheduling

The task scheduling problem consists in, given a set of dependent tasks and its dependencies, choosing in which resource each task will execute. A scheduler, in general, has an objective function. For example, a scheduler can try to minimize the execution time (makespan) of the whole process, maximize the resource utilization or maximize the overall system throughput. We focus on the former.

A workflow is usually represented by a directed acyclic graph (DAG) $G = (V, E)$, where $V$ is the set of tasks to be executed and $E$ is the set of edges that represent data dependencies between tasks. Thus, an edge $e_{i,j}$, with its source on task $t_i$ and its target on task $t_j$, means that $t_j$ can start its execution only after $t_i$ ends its execution and sends all necessary data to $t_j$.

Dependent task scheduling is an NP-Complete problem [3], hence the associated optimization problem is NP-Hard. To deal with this, we developed a heuristic and a dynamic adaptive approach to schedule tasks on grids [1].

In a grid environment, besides the NP-Completeness of the problem, more difficulties arise. The grid is dynamic, the resources are heterogeneous and they have performance variations, since it is a shared environment. In this work we combine some techniques to deal with these characteristics, with the objective of minimizing the makespan of the processes. A four-step scheme representing the scheduling strategy is shown in Figure 1. In the first step, we use the *Path Clustering Heuristic* (PCH) [1] to make the initial schedule of tasks. The *round-based* technique presented in [1] is used in the second step, where the algorithm decides dynamically on each *round* which scheduled tasks will be sent to execution. With this, the algorithm can have performance feedback from the resources to make rescheduling decisions based on the initial schedule given by the PCH algorithm, avoiding to send tasks to execute in resources with poor performance. The dynamic module chooses which tasks will be sent to execution on each round supported by the adaptive module, which regulates the size of each round. This adaptive round size regulation tries to avoid sending too many tasks to execute in resources that had strong performance variations in the past. The round size is measured in terms of execution time of the tasks on a given resource, so it depends on each resource capacity. The execution is the third step, and the rescheduling is the fourth step. We briefly describe these algorithms in Section 4.

In this work we modify the PCH heuristic to search for gaps in the schedule (or queue) of each resource when scheduling new processes. With this, we aim to reduce the overall makespan of all processes without interfering in processes that are already scheduled.

## 3. Related work

Task scheduling is studied in homogeneous systems ([6]) as well as in heterogeneous systems ([8, 11]).

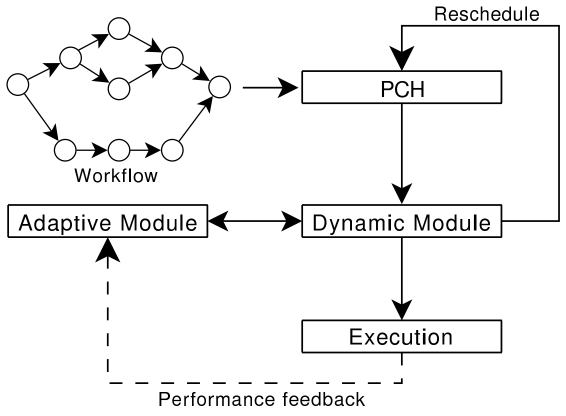

**Figure 1. Four-step scheme.**

Two well known task schedulers for heterogeneous systems are the Heterogeneous Earliest Finish Time (HEFT) [11] and the Critical Path on a Processor (CPOP) [11]. These algorithms schedule dependent tasks on heterogeneous environments without considering performance variations and dynamic behaviour.

A DAG meta-scheduler for grids is proposed in DAGMan [5]. It just sends one task at a time to the scheduler, which schedules the tasks like independent tasks, without knowledge about dependencies.

In [7], a case study on dynamic scheduling for scientific workflow applications on the grid is presented. It proposes a static reschedule with iterations over the workflow application, generating a DAG with the tasks that to be rescheduled on each iteration. After that, the generated DAG is scheduled, actually performing a reschedule of its tasks.

A dynamic two level scheduling for wide area networks is proposed in [2], where a top level scheduler consults the second level schedulers to select in which LAN the process will be executed. There is no way of splitting the process to execute in more than one LAN, what can lead to higher completion times. Also, all schedulers in the second level must reply each scheduling request. If there are many requests, this may overload the schedulers.

Besides many works dealing with workflow scheduling on grids [12], to the best of our knowledge there is no work regarding algorithms for execution of more than one process at the same time on a grid, with considerations of performance losses, dynamicity of the environment and reschedule. Hence, we consider it as an open problem which we focus in this paper.

## 4. Algorithms overview

This section gives an overview of the algorithms used to schedule tasks within the four-step scheme. The algorithms are briefly described and details can be found in [1], where a comparison of PCH with HEFT and CPOP is provided.

## 4.1. PCH

The Path Clustering Heuristic uses the clustering technique to generate groups (clusters) of tasks. The PCH groups paths of the DAG, creating clusters of tasks that reduces the communication between them. Each cluster has tasks that are initially scheduled on the same resource.

First, the algorithm computes for each task some attributes based on informations given by the middleware and by task specifications, defined as follows.

- ***Weight (Computation Cost):***

$$w_i = \frac{instructions_i}{power_r}$$

$w_i$ represents the computation cost of the task $i$ in the resource $r$. $Power_r$ is the processing power of the resource $r$, in instructions per second.

- ***Communication Cost:***

$$c_{i,j} = \frac{data_{i,j}}{bandwidth_{r,t}}$$

$c_{i,j}$ represents the communication cost between tasks $i$ and $j$, using the link between the resources $r$ and $t$, where they are scheduled. If $r = t$ , $c_{i,j} = 0$.

- ***Priority:***

$$P_i = \begin{cases} w_i, & \text{if } i \text{ is the last task} \\ w_i + \max\limits_{t_j \in succ(n_i)} (c_{i,j} + P_j), & \text{otherwise} \end{cases}$$

$P_i$ is the priority level of the task $i$.

- ***Earliest Start Time:***

$$EST(t_i, r_k) = \begin{cases} Time(r_k), & \text{if } i = 1 \\ \max\{Time(r_k), EST_{pred}\}, & \text{otherwise} \end{cases}$$

$EST(t_i, r_k)$ represents the earliest start time of the task $i$ in resource $k$. $Time(r_k)$ is the time when the resource $k$ is ready for task execution, and $EST_{pred} = \max\limits_{t_h \in pred(t_i)} (EST_h + w_h + c_{h,i})$.

- ***Estimated Finish Time:***

$$EFT(t_i, r_k) = EST(t_i, r_k) + \frac{instructions_i}{power_k}$$

$EFT(t_i, r_k)$ represents the estimated finish time of the task $i$ in the resource $k$.

Then, the algorithm uses the priority to select the first task to be added to the first cluster (clustering phase). The first node (or task) $n_i$ selected to compose a cluster $cls_k$ is the unscheduled node with the highest priority. It is added to $cls_k$, then the algorithm starts a depth-first search on the DAG starting on $n_i$, selecting $n_s \in succ(n_i)$ with the highest $P_s + EST_s$ and adding it to $cls_k$, until $n_s$ has no unscheduled successors.

For each cluster created, the algorithm selects a resource to schedule it. The processor selection step is performed after each clustering step. The algorithm creates a cluster, selects a processor to the created cluster, recalculates the nodes attributes and repeats these steps until all nodes are scheduled.

The criterion to choose the processor to a cluster is to minimize the EST of the successor task of the cluster being scheduled. To accomplish this, the first step is to calculate the EFT of each node of the cluster on each available resource. If the current resource already has a cluster of the same DAG, the tasks are sorted in descending order of priority to obey the precedence constraints, avoiding deadlocks. The next step is to calculate the EST of the successor of the last node of the cluster being scheduled. The first cluster has no predecessors and no successors, since it starts on the process' first node and ends on the last one. So, having no successors, it is scheduled on the resource that gives the smallest EFT for its last node. In short, a cluster $cls_k$ is scheduled on the resource that gives the smallest EST for the successor of $cls_k$. After the scheduling of each cluster, the tasks attributes are recomputed.

## 4.2. Dynamic PCH

The dynamic PCH introduces dynamic scheduling behaviour in the PCH algorithm. The initial schedule made by the PCH algorithm is sent to the dynamic module and it decides which tasks will be executed based on the current round of execution. The number of rounds is determined when the scheduling starts.

After the initial schedule made by the static PCH, the dynamic schedule is started. First, the algorithm selects only part of the DAG to be sent to execution. A node $n$ is sent to execution in a round $k$ (of a total of $T$ rounds) if $n$ has not started its execution and if $EFT_n \leq \frac{makespan}{T/k}$, with $1 \leq k \leq T$, or if the task is the first task on the resource schedule. As the tasks are being executed, the scheduler can compare the real execution time with that estimated by the tasks' attributes calculated by the algorithm. If there is a performance loss higher than a threshold (for example, 10%) in a resource, the tasks that have not started their execution and are scheduled to that resource are rescheduled using the PCH algorithm. The rounds are repeated until all tasks are sent to execution.

### 4.3. Adaptive extension

After providing a dynamic algorithm to deal with performance losses in resources, now we show an adaptive algorithm that adjusts the number of rounds to DAGs of different sizes and resources with variable performance. The objective of this adaptive extension is to vary the round size according to the weight of the tasks, the size of the process and the resources performance. Some characteristics are desirable in the adaptive algorithm: Adapt the size of the rounds according to the weight of the tasks, adapt the size of each round according to the performance of each resource, and consider the performance history of each resource when deciding the size of the round.

With this, we developed a mathematical model that shows how these characteristics can be adopted in the scheduling algorithm. The dynamic module is now supported by the adaptive module. The size of the initial round depends on the size of the tasks on the execution queue, and the next round sizes depend on information about resources performance provided by the mathematical model formulated in the adaptive extension. An overview of how the whole scheduling algorithm works is shown in Algorithm 1.

**Algorithm 1** Algorithm Overview

```
1:  Schedule the DAG using the PCH
2:  Select tasks to send to execution on each resource
3:  Send tasks of current round to execution
4:  while there are tasks not sent to execution do
5:      $t \Leftarrow$ wait_next_task_to_finish() //blocking wait
6:      $r \Leftarrow$ resource of $t$
7:      if $t$ is the last task of this round on $r$ then
8:          if $total\_exec\_time_r > expected\_exec\_time_r + threshold$ then
9:              Reschedule not started tasks of $r$ with PCH
10:             $r \Leftarrow$ new resource to remaining tasks of $r$
11:         end if
12:         Calculate the round size on $r$ using performance information provided by the adaptive module
13:         Send tasks of the current round to execute on $r$
14:     end if
15: end while
```

## 5. Gap searching algorithm

In this section we present an algorithm to queue tasks on the schedule of each resource considering gaps between the already scheduled tasks. To make it simple, we modified the original PCH to be more flexible. The modification was done to give simpler gaps searching algorithms and to avoid deadlocks between processes.

The modification on the PCH is in the depth-first search that composes the clusters. Instead of stopping the search when it finds a task that has no unscheduled successors, the modified PCH stops when it finds a task $t_i$ that has no scheduled predecessor, and $t_i$ is not included into the cluster. With this, the clusters always have only tasks with all predecessors already scheduled. Also, the clusters are potentially smaller than with the original PCH, what increases the probability of finding a gap with enough space to each cluster. On the other hand, smaller the clusters, higher the number of searches for gaps. So, finding gaps for one task at a time may be too costly, thus using small size clusters is an intermediary solution. Hereafter, *PCH* refers to this modified version and *original PCH* refers to the version presented in Section 4.1.

Starting the scheduling of a process, if there are no processes currently scheduled on the available resources, the scheduler does not need to search for gaps, so the scheduling proceeds following the dynamic PCH algorithm with the adaptive extension. If there are tasks currently assigned to one or more resources, the algorithm searches for gaps in the schedule to place each cluster.

Let the schedule (queue) of the resource $r$ be $S_r = \{t_1, t_2, ..., t_k\}$. The gap $g_{c,r}$ for the cluster $c$, $c = \{t_1^c, t_2^c, ..., t_m^c\}$, in the resource $r$, is defined as:

$$g_{c,r} = \min_{t_j \in S_r} (j)$$

such that $(EST(t_{j+1}, r) - EFT(t_j, r)) \times s\_margin > size_{c,r}$ and $EST(t_{j+1}, r) - EST(t_1^c, r) > size_{c,r}$.

We call *security margin*, $s\_margin$, a free space in the found gap to give room for possible lost of performance in resources. This way, if the resource performance is worst than expected, the cluster inserted in the gap can execute with minor interference in the tasks previously scheduled on that resource. This security margin is relative to the size of the gap. For example, if we want $10\%$ of security margin, we use $s\_margin = 0.9$. Also, we define the size of a cluster $c$ on $S_r$ as the difference between the EFT of the last task of the cluster and the EST of the first task of the cluster on $r$, or, $size_{c,r} = EFT(t_m^c, r) - EST(t_1^c, r)$.

To avoid deadlocks, one verification is made when the gap searching algorithm finds a gap for a cluster. After composing each cluster, the algorithm generates a set of dependent tasks for each task in the cluster. The set of dependent tasks of a task $t_i$, $D_{t_i}$, is composed of all tasks in any path from $t_i$ to $t_k$ (the last task of the process). Before assigning a gap to a cluster $c$, the algorithm verifies if there are no tasks ahead of the gap in the schedule which $t_m^c$ (the last task of the cluster) depends on. If there is such a task, the gap cannot be assigned to that cluster, and the algorithm proceeds with the search. The gap searching algorithm is shown in Algorithm 2.

**Algorithm 2** search_gap($S_r$, $c$)

```
1:  $size_{c,r} \Leftarrow EFT(t^c_m, r) - EST(t^c_1, r)$
2:  $k \Leftarrow$ number of tasks in $S_r$
3:  $i \Leftarrow 1$
4:  if $(EST(t_1, r) \times s\_margin) > size_{c,r}$ then
5:      Compute ESTs and EFTs for $t^c_j \in c$ on the current gap
6:      $D_{t^c_m} \Leftarrow$ tasks ahead which $t^c_m$ depends on
7:      if $(EST(t_1, r) - EST(t^c_1, r) > size_{c,r})$ and $D_{t^c_m} = \emptyset$ then
8:          $g_{c,r} = 0$; return $g_{c,r}$
9:      end if
10: end if
11: for $i = 1$ to $k - 1$ do
12:     if $((EST(t_{i+1}, r) - EFT(t_i, r)) \times s\_margin) > size_{c,r}$ then
13:         Compute ESTs and EFTs on current gap $\forall t^c_j \in c$
14:         $D_{t^c_m} \Leftarrow$ tasks ahead which $t^c_m$ depends on
15:         if $(EST(t_{i+1}, r) - EST(t^c_1, r) > size_{c,r})$ and $D_{t^c_m} = \emptyset$ then
16:             $g_{c,r} = i$; return $g_{c,r}$
17:         end if
18:     end if
19: end for
20: $g_{c,r} = k$; return $g_{c,r}$ //no gap found
```

The gap searching algorithm first determines the size of the cluster $c$ being scheduled and the number of tasks in the schedule of $r$ (lines 1 and 2). Then it verifies if the first task on the resource has $EST > 0$ and if $c$ is smaller than this gap in the start of the schedule, considering the security margin (line 4). If $c$ fits in the gap, the algorithm verifies if there is room for the cluster discounting its EST, since we do not want to interfer on the execution of tasks already in the schedule, and it verifies if there is a deadlock (line 6). If there is no room for the cluster in the beginning of the schedule or there is deadlock, the algorithm starts to iterate over the tasks on the schedule (line 11). The algorithm searches a gap between the task on the current position, $t_i$, and the next task, $t_{i+1}$ (lines 12 to 18). If no gaps were found, the position is set to the last one (line 20).

The processor selection algorithm with gap searching is shown in Algorithm 3. For all resources available, the algorithm searches for gaps (line 2) and inserts the cluster $c$ in the position returned by the gap searching algorithm (line 3). The insertion is done after the task on the stated position, starting with $t_1 \in S_r$ in position 1. In line 5 the algorithm computes the EST of the successor of $c$. Finally, the algorithm returns the resource with the smaller EST for $c$'s successor in the DAG (line 7).

An example is shown in Figure 2. Two process, $P0 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\}$ and $P1 = \{t_A, t_B, t_C, t_D, t_E, t_F\}$, are scheduled with gap searching. For the example, consider that resources $R0$ and $R1$ have same performance. The first cluster of $P0$ scheduled is $cls_0 = \{t_1, t_2, t_3, t_4, t_6\}$. Then $cls_1 = \{t_5, t_7\}$ is scheduled. Finally, $cls_2 = \{t_8, t_9\}$ is scheduled. After scheduling $P0$, $P_1$ is scheduled, starting with $cls_0 = \{t_A, t_B, t_D\}$ on the gap found before $t_5$. Then, $cls_1 = \{t_C, t_E\}$ is scheduled on the gap between $t_6$ and $t_8$. Finally, $cls_2 = \{t_F\}$ is scheduled after $t_E$ and before $t_8$.

**Algorithm 3** get_best_resource($c$)

```
1: for all $r$ in $resources$ do
2:     $g_{c,r} \Leftarrow$ search_gap($S_r$, $c$)
3:     $schedule \Leftarrow$ Insert $cluster$ on $S_r$ in position $g_{c,r}$
4:     calculate_EFT($t^c_m$);
5:     $time_r \Leftarrow$ calculate_EST(successor($t^c_m$))
6: end for
7: return resource $r$ with the smallest $time_r$
```

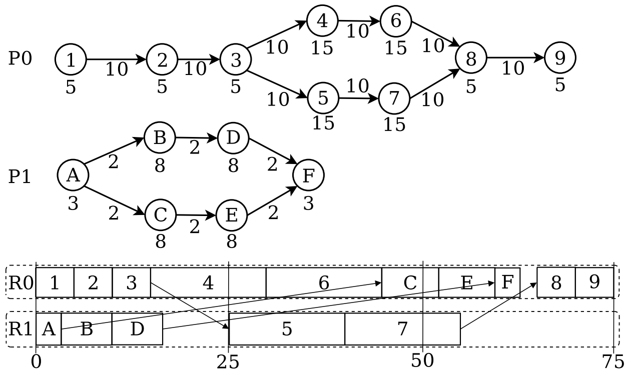

**Figure 2. Two DAGs and resulting schedule.**

In the example, $EST(t_C, R0) = 45$ and $EFT(t_E, R0) = 61$. Also, $EFT(t_6, R0) = 45$ and $EST(t_8, R0) = 65$. Tasks $t_C$ and $t_E$ can be scheduled on the gap between $t_6$ and $t_8$ considering a security margin of at most 20% or, more precisely, $s\_margin > 1 - \frac{4}{19}$, since $size(\{t_6, t_8\}, R0) = 19$ and $EST(t_8, R0) - EFT(t_6, R0) = 4$.

## 5.1. Rescheduling of Tasks

When a resource does not perform as expected, the dynamic algorithm allows to change the current schedule to avoid sending tasks to that resource. To do this, the algorithm reschedules the not executed tasks that are in the schedule of the resource. When there is only one process scheduled on a set of resources, the rescheduling can be made by reallocating the tasks originally on the resource with poor performance. In this case, since there are only tasks of one process on all resources, the selection of a new resource demands simple verifications to avoid deadlocks. When more than one process share the same set of resources and they also use gaps in the schedule of other processes, the rescheduling of tasks becomes more complex. When tasks are reallocated, there is a possibility of deadlocks between tasks of different processes, what we call an *inter-process task deadlock*.

Although complex algorithms could be developed to handle this problem, in this paper we deal with it in a simple way. If the dynamic algorithm measures a poor performance of a resource at the end of the round, the algorithm searches a new resource for each cluster ahead of the end of the current round on the aforesaid resource. But, if there is no gap for a cluster and it is rescheduled on the end of the current schedule of a resource, deadlocks can also occur. To avoid deadlocks, the proposed algorithm sorts all tasks in the schedule by their EST when rescheduling. This way, a task will never be scheduled before one of its predecessors or after one of its successors. Note that there is no gap searching in the rescheduling. When sorting tasks by their EST, the scheduler tries to mantain the rescheduled cluster starting at a time near that in the original resource, and at the same time it tries to minimize the cluster's end time. An overview of the rescheduling is shown in Algorithm 4.

**Algorithm 4** reschedule($S_r$)

```
1:  $C_{res} \Leftarrow$ clusters of $r$ to reschedule
2:  for all $c \in C_{res}$ do
3:      for all $r$ in $resources$ do
4:          $schedule \Leftarrow$ Insert $cluster$ on $S_r$
5:          Order $S_r$ by EST
6:          calculate_EFT($t^c_m$);
7:          $time_r \Leftarrow$ calculate_EST(successor($t^c_m$))
8:      end for
9:  end for
10: return resource $r$ with the smallest $time_r$
```

## 6. Experimental Results

In this section we evaluate the initial makespan of the processes scheduled with the gap searching algorithm and the makespans when executing and rescheduling the processes on resources with performance variation. The comparison uses two processes (process zero and process one). The scheduling with gap searching is compared with scheduling without gap searching, where the first process scheduled executes all its tasks scheduled on a resource before the next process can execute on the same resource. With this, the second process can be scheduled on the remaining free resources or on resources used by the first process, but only after the tasks of the first process.

In this work we consider that grids are composed of groups of resources. For example, a LAN or a cluster could be a group. The resources inside the group have same link capacities between them, and these capacities can be different on each group. The connections between groups are heterogeneous.

Fifteen graphs supported by the Xavantes programming model were randomly taken for the experiments, with number of nodes between 7 and 82. Medium communication means that the communication and computation costs were randomly generated in the same interval (from 500 to 1100 time units). High communication means all communication costs (from 1100 to 1600 time units) were higher than all computation costs (from 500 to 1100 time units). The experiments were made simulating a group topology, as described in the *Infrastructure* section. For each number of groups, varying from 2 to 25, two graphs were scheduled 15000 times. Each group had a random number of heterogeneous resources, varying from 1 to 5.

For each execution, two processes among the fifteen graphs were randomly taken, which means $15 \times 15 = 225$ different configurations of process scenarios. First, process zero were scheduled using the PCH algorithm. Then, process one was scheduled with and without the gap searching algorithm. We used a security margin of 10% on the experiments ($s\_margin = 0.9$).

The main comparison metrics used in the literature are the Schedule Length Ratio (SLR) and the speedup. The SLR shows how many times the schedule length (makespan) is bigger than the execution of the critical path of the DAG on the best resource available (less is better). The speedup shows how many times the makespan is smaller than the makespan of all tasks in sequence on the best resource (more is better).

Figure 3 shows the average SLR for process one in the initial schedule (before execution) with and without the gap searching for medium communication. Fewer the number of groups, higher the SLR for both algorithms, since there are less options of resources. But, fewer the number of groups, higher the gain with the gap searching algorithm when compared to scheduling without gap searching. This is because process zero was scheduled on the best resources, leaving gaps on its schedule. So, when process one is scheduled, there are less options of free resources, and the free resources currently have worse performance than the not free ones, what makes the gaps good options to schedule process one's clusters. On the other hand, with more groups, although the algorithm with gap searching still performing better, the difference on the SLR is smaller than with a few groups. Thus, even with many options of resources (higher number of groups), the gap searching algorithm can find gaps that worth to be used in the scheduling of process one. This analysis is also valid for the speedup results

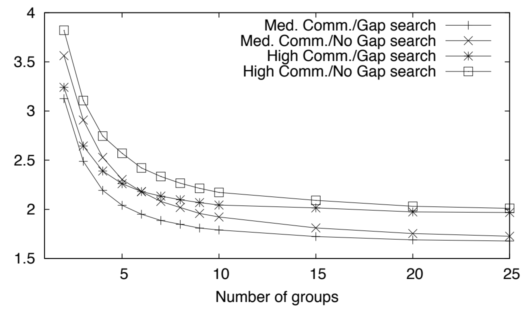

**Figure 3. Average SLR for P1.**

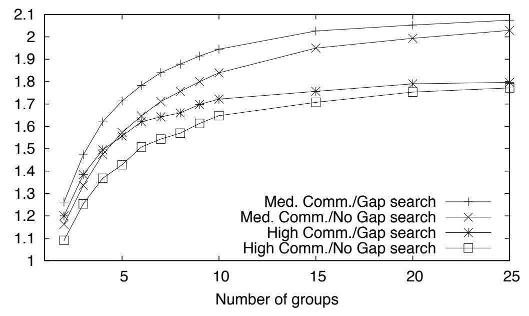

**Figure 4. Average speedup for P1.**

with medium communication (Figure 4), strengthening the conclusions. Note that the initial schedule does not interfer on process zero, so its makespan is the same with both algorithms. Thus, it is not shown on inital schedule results.

In a high communication scenario, the SLR results (Figure 3) show the same behaviour of SLR for medium communication. The graphic is slightly different from medium communication ones, since the SLR differences are wider with a few groups when compared to differences with medium communication. This is because scheduling on a few resources supresses the high communication costs, resulting in many tasks of the same process on the same resource when using the gap searching algorithm. On the other hand, more resources can spread the tasks, increasing the communication costs and approximating the results of both algorithms. For the speedup results with high communication (Figure 4), we can observe a similar behaviour, but with a smaller difference when having a few groups.

Figure 5 compares the average of speedups for the initial schedule when scheduling three processes, namely $P0$, $P1$ and $P2$, with medium communication. Naturally, when there are three processes scheduled, the speedup of $P2$ is worse than speedup of $P1$, but note that the gain with the gap searching algorithm is bigger to $P2$ than to $P1$. We also observed a similar behaviour for SLR in this scenario.

After evaluating the initial schedule, we now evaluate how the initial schedules behave when executing the tasks.

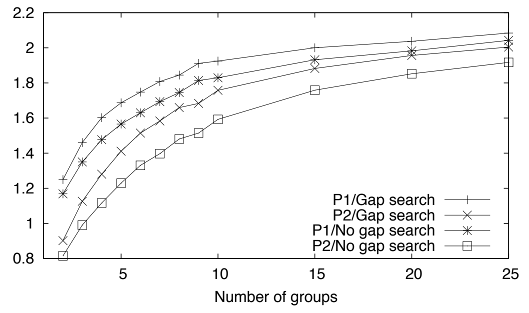

**Figure 5. Average speedups for P1 and P2.**

No rescheduling means that all tasks are executed in the resources given by the initial schedule, no matter their performance, and rescheduling means that the tasks are rescheduled using Algorithm 4. Note that there is no *reallocation* of tasks, so there is no overhead of moving tasks, since the reschedule is made before the tasks are sent to execution. To determine the resources performance, we first generated a pattern of resources performance based on measurements made on computers in our laboratories. With this, in the beginning of the simulation, we generated processing powers for each time interval on each resource. These processing powers were the same for the execution of each algorithm. The results show the average over both processes with high communication.

Figure 6 shows the SLR results for the execution of process zero and one with and without gap search, and with and without rescheduling. The executions without reschedule have similar results. This means that in an environment with performance variation, the gap searching algorithm can be useless if there is no rescheduling policy. This is because all the gain in the gaps is lost when the performance drops in the resources. This loss of performance works like a snowball when there are no gaps in the schedule, affecting all tasks of both processes. On the other hand, when the gaps are empty, there is room to “absorb” the delays in task execution, and this delay does not affect the processes in a way like it does when there are no gaps in the schedule. When the rescheduling algorithm is used, there is a visible improvement in the schedule.

Comparing the reschedule with and without gap search we can see that the gap search can improve the results when there are a few groups. When there is a higher number of groups (more than 10), the results of rescheduling with and without gap searching are equivalent. This can be explained by the fact that we use the same algorithm in both cases, and this algorithm has no gap searching because of potential deadlocks. Thus, when there are many options of resources, the gaps used in the initial schedule have less impact on the final execution time. The gain with the gap searching algorithm is due to the initial gain in the schedule and the security margin, which can "absorb" some performance loss. Hence, the gap searching algorithm is very useful when there is a little or no performance loss, and can make little better schedules when there is considerable performance loss by using a simple rescheduling algorithm.

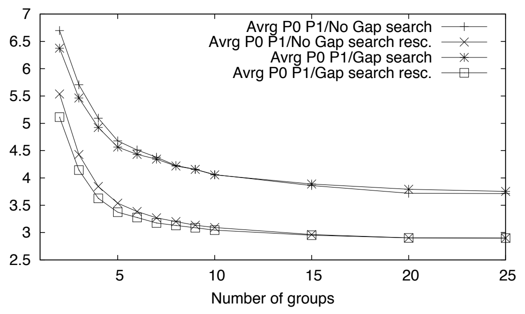

**Figure 6. SLR for execution of P0 and P1.**

The previous analysis is also valid for the speedup results (Figure 7), except for results with a few groups. In this case, because the algorithm has no good options to make the reschedule, the sorting of tasks on the available resources seems to worsen the schedule. However, this problem is not seen when the number of groups grows to more than 4.

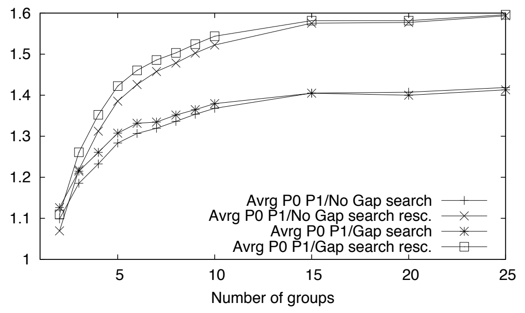

**Figure 7. Speedup for execution of P0 and P1.**

## 7. Conclusion

In this work we study how to deal with performance issues when scheduling workflows on the grid. We propose an algorithm to schedule more than one process using the same set of resources by searching for gaps in the current schedule. This gap searching algorithm uses a security margin to prevent interference on processes already on the schedule, and it is combined with a rescheduling algorithm.

The experimental results suggest that the gap searching algorithm in conjunction with the rescheduling algorithm can provide good schedules in situations inherent to the grid. While the gap searching algorithm, supported by the dynamic and adaptive modules, tries to make good schedules when resources have good performance (with little or no variation in expected performance), the rescheduling deals with heavier performance variations.

Future works include the development of an advanced rescheduling algorithm and a performance prediction model to interact with the dynamic and adaptive modules. It also could provide information to the security margin in the gaps. This can contribute to avoid rescheduling of tasks on foreseeable situations of resources load.

## References

[1] L. F. Bittencourt and E. R. M. Madeira. A performance oriented adaptive scheduler for dependent tasks on grids. *Concurrency and Computation: Practice and Experience (to appear)*.

[2] H. Chen and M. Maheswaran. Distributed dynamic scheduling of composite tasks on grid computing systems. In *11th IEEE Heterogeneous Computing Workshop*, pages 119–128, Washington, DC, USA, 2002. IEEE Computer Society.

[3] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, 1995.

[4] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[5] J. Frey. Condor DAGMan: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/, 2002.

[6] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[7] R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694, New York, NY, USA, 2005. ACM Press.

[8] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *13th Heterogeneous Computing Workshop*, pages 111,123. IEEE Computer Society, 2004.

[9] X.-H. Sun and M. Wu. Grid harvest service: A system for long-term, application-level task scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 25.1, Washington, DC, USA, 2003. IEEE Computer Society.

[10] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors. *Workflows for e-Science. Scientific Workflows for Grids*. Springer Verlag, 2007.

[11] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel and Distributed Systems*, 13(3):260–274, 2002.

[12] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.