A Dynamic Approach for Scheduling Dependent Tasks on the Xavantes Grid Middleware

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

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

ABSTRACT

A scheduler must consider the heterogeneity and communication delays when scheduling dependent tasks on a grid. The task scheduling problem is NP-Complete in general, what led us to the development of a heuristic for the associated optimization problem. In this work we present a dynamic approach to schedule dependent tasks onto a grid based on the Xavantes grid middleware. The developed dynamic approach is applied to the *Path Clustering Heuristic* (PCH), and introduces the concept of *rounds*, which take turns sending tasks to execution and evaluating the performance of the resources. The experiments show that the *round-based* dynamic schedule can minimize the effects of performance losses while executing processes on the grid.

Categories and Subject Descriptors

D.1.4 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Algorithms

Keywords

Task scheduling, grid computing, workflow

1. INTRODUCTION

Grid computing has emerged as a distributed high-power environment, where any kind of computational resource can be part. Thus, it is a heterogeneous computing platform that has different link and processor capacities. Furthermore, a grid is in general not dedicated, what leads to load variations in links and processors. This dynamic behavior is responsible for many issues in a grid, including ones concerning the task scheduling problem.

An application (or process) may have many tasks to execute, each one with its role in the process completion. During the process execution a task may need data computed by other task(s) of the same process, what is called *task dependency*. Hence, a process is composed of tasks and its dependencies, constituting a *workflow*, where tasks that have no dependency among them can execute in parallel.

In this paper we propose an approach to schedule processes composed of dependent tasks in the dynamic environment of a grid. The approach is applied to the *Path Clustering Heuristic* algorithm (PCH) [2], the Xavantes grid middleware [3] scheduler. The approach is experimentally evaluated and it shows improvement over a static scheduler.

Section 2 gives a brief introduction to the Xavantes grid middleware, and Section 3 shows some related works. The dynamic approach is explained in Section 4, in conjunction with the PCH algorithm. Simulations are shown in Section 5, and section 6 ends the paper with the conclusions.

This is a pre-print version.
The final version is available at the publisher's website.

2. XAVANTES MIDDLEWARE

Xavantes enacts workflow-like processes, differently from most grid middlewares, which only support a bag of independent tasks. It is composed of a programming model and a support infrastructure, which enacts the specified process in a grid environment. For details please refer to [3].

Xavantes' programming model allows to specify applications as structured processes, containing a hierarchy of control constructs to determine the control flow. In a process, an activity represents an atomic task, and a *controller* represents a control structure that organizes the execution order of inner process elements. The controllers can be nested, allowing the hierarchical specification of the control, similarly to structured programming languages.

A *Directed Acyclic Graph* (DAG) is used to represent a process: nodes are tasks, labeled with computation costs; edges are dependency constraints, labeled with communication costs. In the DAG of Figure 1, the rectangle 1 represents a parallel controller containing the tasks 7 and 8, and the rectangle 4 represents a sequential controller containing the activities 2, 5 and 10, and the controller 1.

A controller knows the execution state of the tasks subordinated to it. Hence, controllers can provide recovery, dependability, and can facilitate the fault tolerance mechanism, while they scale well, distributing the process management. In addition, controllers have a shared memory, which could be used to allocate shared variables, making possible the communication between parallel tasks.

Xavantes organizes the grid in autonomous groups of distributed resources. For example, a cluster or a LAN could be a group. In each group there are three kinds of entities:

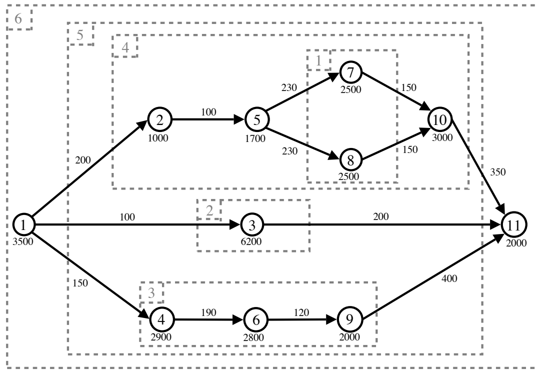

Figure 1: Task graph example.

one *Group Manager* (GM), one or more *Process Managers* (PM) and one or more *Activity Managers* (AM). The GM is responsible for mantaining a repository of resources that are in the group, with informations about bandwidth, processing power, load, etc. These informations are dynamic and can be updated by the infrastructure when necessary. The PMs are responsible for executing the controllers and managing the execution of processes, and the AMs are the workers that execute tasks managed by the PMs.

The main goals of Xavantes are high-performance and reliable enactment of applications. The hierarchical execution of structured processes in distributed groups of resources leverages the scalability of the grid. The explicit support to process execution allows better fault tolerance and scheduling, improving the grid reliability and performance. Also, the concept of controllers provides distributed management of processes, giving an efficient recovery mechanism. The development of a good task scheduler for the Xavantes is fundamental to deploy a middleware that provides a recovey background with efficient process execution.

3. RELATED WORK

Task scheduling is studied in homogeneous systems ([8]) as well as in heterogeneous systems ([1, 10]). List scheduling, clustering and task duplication are DAG scheduling techniques for homogeneous and heterogeneous systems.

The Heterogeneous Earliest Finish Time (HEFT) [10] algorithm uses the list scheduling technique. First, HEFT computes the $rank_u$ value for each task, traversing the graph backwards. The unscheduled task with the highest $rank_u$ value is scheduled on the processor that gives the smallest estimated finish time for the task. HEFT looks for idle time slots in the current schedule when scheduling a task. The time complexity of HEFT is $O(rn^3)$ [7], where r is the number of resources and n is the number of tasks of the process. The Critical Path on a Processor (CPOP) [10] algorithm is similar to HEFT. The main difference is that all nodes on the DAG's critical path are scheduled on the same processor.

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 [9], 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 static DAG with the tasks that would be rescheduled on each iteration. After that, the generated DAG is scheduled, actually performing a reschedule of its tasks.

The PCH algorithm [2] uses a hybrid clustering-list-scheduling strategy to schedule task graphs on a grid. HEFT, CPOP and other grid task schedulers ([5], [4], [9], [6]) do not consider the existence of controllers, entities used by the Xavantes middleware to control the process execution and provide recovery. This can lead to the dispersal of the application tasks, hence expensive communication costs. PCH is a heuristic that schedules related tasks and controllers on nearby resources, minimizing their completion times.

4. A DYNAMIC APPROACH

The objective of a task scheduler is to minimize the execution time (*makespan*) of a process by distributing its tasks among the resources. In Xavantes, controllers have an important role in the process execution, since they are responsible for managing tasks execution and communication, being able to provide efficient recovery. So, a communication between two tasks must be via tasks' controllers, giving to the controllers the knowledge to make the recovery of tasks when necessary. Combining the objective of a task scheduler with the exigence of communications to be via controllers, the scheduler objective is straightforward: minimize the makespan without creating too much communication overhead between tasks and controllers.

The static PCH gives good results in a static heterogeneous environment with a mesh heterogeneous (no-group) topology [2]. In a dynamic heterogeneous environment there is a need for a dynamic approach, adapting the resource selection to the dynamic behavior of the grid. We developed an approach to dynamically schedule workflows, trying to minimize the effects of a possible performance loss.

A dynamic task scheduler can use, in general, two strategies: schedule all tasks statically and then reschedule tasks when necessary; or gradually schedule and execute the tasks based in a priority policy. We have chosen the first strategy because it follows a process FCFS (First Come, First Served) policy. The PCH make the initial schedule and then a *round-based* approach is applied to reschedule tasks when necessary. Algorithm 1 is an overview of this approach.

Algorithm 1 Dynamic Approach Overview

1: Schedule DAG G using the static PCH Algorithm
2: **while** not(all nodes of G have finished) **do**
3: Select tasks to execute according to a policy.
4: Send tasks of this round to execution.
5: Evaluate the resources performance.
6: Reschedule tasks if necessary.

To make a dynamic schedule of tasks we developed the concept of *rounds*. In each round some tasks are selected based on a criterion and sent to execution. Then, the scheduler verifies the performance obtained on each resource used on that round. If the performance of a resource is below a threshold, the algorithm reschedules the non executed tasks.

In Xavantes, the scheduler is responsible for detecting and rescheduling tasks that are on resources with low performance. Fails in resources are detected by the middleware and the tasks are sent to the scheduler, so it can reschedule them. The process of recovery is handled by the middleware, using the information provided by the controllers.

4.1 Static PCH

The first step of the dynamic scheduling is to schedule the tasks using the static PCH. To reduce the communication between tasks and controllers, PCH groups paths of the DAG, creating *clusters* of tasks. The creation of groups based on paths of the DAG has a good impact on minimizing the controllers' communication overhead, since the created clusters have non parallel tasks of a controller, permitting the controller to execute concurrently in the same resource of most of its tasks. For example, in Figure 1 the tasks 3, 6, and 9 are under the controller 3. This way, if tasks 3, 6, and 9 execute in the same resource of controller 3, the communication cost between these tasks would be 0. Note that any communication between entities executing in the same resource is 0 because the data does not cross any link.

4.1.1 Definitions

The PCH algorithm uses some graph attributes, calculated as follows.

- ***Weight (Computation Cost):*** $w_i = \dfrac{instructions_i}{power_r}$

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

- ***Communication Cost:*** $c_{i,j} = \dfrac{data_{i,j}}{bandwidth_{r,t}}$

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

- ***Priority:*** $P_i = w_i + \max\limits_{n_j \in succ(n_i)} (c_{i,j} + P_j)$

P_i is the priority level of the node i. The priority of the exit node is $P_{exit} = w_{exit}$.

- ***Earliest Start Time:***

$$EST(n_i, r_k) = \max\{Time(r_k), \max_{n_h \in pred(n_i)} (EST_h + w_h + c_{h,i})\}$$

$EST(n_i, r_k)$ represents the earliest start time of the node i in resource k. $Time(r_k)$ is the time when the resource k is ready for task execution.

- ***Estimated Finish Time:***

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

$EFT(n_i, r_k)$ represents the estimated finish time of the node i in the resource r_k.

The initial values of the attributes are calculated assigning each task to a different processor in a virtual homogeneous system with an unbounded number of processors. These processors have the power of the best processor available in the real system, and all links have the highest bandwidth available in the real system. After the calculation, while there are unscheduled nodes, the algorithm creates a cluster of tasks (Section 4.1.2) and selects a processor to it (Section 4.1.3). The necessary information (e.g. available processing power and bandwidth) is given by the infrastructure. Finally, the algorithm schedules the controllers (Section 4.1.4), trying to minimize their communication costs.

4.1.2 Task Selection and Clustering

In this step of the algorithm, the heuristic selects tasks to compose clusters. Tasks that are on the same cluster will be initially scheduled on the same processor. 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 $n_s \in suc(n_i)$ with the highest $P_s + EST_s$ and adding it to cls_k, until n_s has no unscheduled successors. Algorithm 2 gives an outline of this strategy.

Algorithm 2 get_next_cluster

1: $n \Leftarrow$ unscheduled node with highest Priority
2: $cluster \Leftarrow cluster \cup n$
3: **while** (n has unscheduled successors) **do**
4: $\quad n_{succ} \Leftarrow successor_i$ of n with highest $P_i + EST_i$
5: $\quad cluster \Leftarrow cluster \cup n_{succ}$
6: $\quad n \Leftarrow n_{succ}$
7: **return** $cluster$

For the graph of our example (Figure 1) and the resources in Table 1, the first node added to the first cluster, cls_0, is n_1, then n_2, n_5, n_7, n_{10} and n_{11}. Then cls_0, which contains the critical path of the initial DAG, is scheduled according to the processor selection strategy and the ESTs, EFTs and Weights are recomputed. After that, the unscheduled node with highest priority is n_4, which is added to cls_1. Then, n_6 and n_9 are added to cls_1. Because of the structure of the task graphs generated by the programming model, where fork and join are paired, each cluster has only one successor task. The other clusters are $cls_2 = \{n_8\}$ and $cls_3 = \{n_3\}$.

Table 1: Resources used in the example. The bandwidth between each resource is given.

Resource	Power	Bandwidth			
0	133	∞	10	5	5
1	130	10	∞	5	5
2	118	5	5	∞	10
3	90	5	5	10	∞

4.1.3 Processor Selection

The processor selection step is realized 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 successors, so it is scheduled on the resource that gives the smallest EFT for its last node. The other clusters have only one successor and this successor is already scheduled, since, by construction, the last node of the cluster does not have unscheduled successors. A cluster cls_k is scheduled on the resource that gives the smallest EST for the successor of cls_k. After the cluster is scheduled, the Weights, ESTs and EFTs are recomputed. For the example of Figure 1, cls_0 is scheduled on resource 0, cls_1 on resource 1, and cls_2 on resource 0, with n_8 before node n_{10}, obeying the precedence constraints. Finally, cls_3 is scheduled on resource 2.

Algorithm 3 get_best_resource

1: **for all** r in $resources$ **do**
2: $\quad schedule \Leftarrow$ Insert $cluster$ in $schedule_r$
3: $\quad$ calculate_EFT($schedule$);
4: $\quad time_r \Leftarrow$ calculate_EST(successor($cluster$))
5: **return** resource r with the smallest $time_r$

4.1.4 Controller Scheduling

With all nodes scheduled, each controller must be assigned to a processor that minimizes the communication with its nodes and with its subcontrollers. Since all tasks communication must be via their controllers, the policy of creating clusters based on sequential tasks in the DAG reduces the controllers communication overhead. A controller can be selected to be scheduled if it has no unscheduled subcontrollers. A controller is scheduled on the resource that minimizes its communication with tasks and other controllers. The whole schedule algorithm is shown in Algorithm 4. The final schedule for the example is shown in Figure 2.

Algorithm 4 PCH Algorithm

1: Assign the DAG to the homogeneous virtual system
2: Compute all tasks attributes
3: **while** there are unscheduled nodes **do**
4: $\quad cluster \Leftarrow$ get_next_cluster()
5: $\quad resource \Leftarrow$ get_best_resource($cluster$)
6: $\quad$ Schedule $cluster$ on $resource$
7: $\quad$ Recalculate Weights, ESTs and EFTs
8: schedule_controllers()

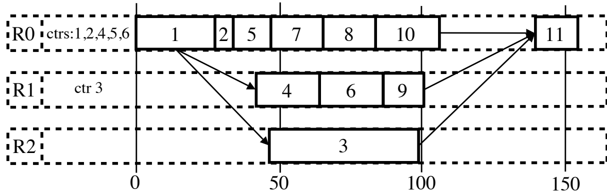

Figure 2: Schedule obtained for the task graph of Figure 1.

4.2 Rounds and Dynamic Reschedule

After the initial schedule, the dynamic reschedule 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 R rounds) if n has not started its execution and if $EFT_n \leq \frac{makespan}{R/k}$, with $1 \leq k \leq T$. As the tasks are being executed, the controllers know the real execution times of each terminated task, so the scheduler can compare the real execution time with that estimated by the tasks attributes calculated by the algorithm. Trying to not leave a resource idle, the scheduler does not wait the last task of the resource to finish. Instead, when a resource ends the execution of the penultimate node of a process scheduled to it, the algorithm calculates the average execution time of the finished tasks on that resource. Then the algorithm compares this real average with the expected one to know if there must be a reschedule on that resource. If there is a performance loss higher than a threshold (for example, 10%) in a resource, the not executed tasks scheduled to that resource are rescheduled using the PCH algorithm. Also, a new resource processing power is set in the repository (GM), accordingly to the measured performance. The dynamic PCH is shown in Algorithm 5.

Algorithm 5 Dynamic PCH Algorithm

1: Schedule DAG G using the PCH Algorithm
2: $R \Leftarrow$ number of rounds
3: **for all** res that has a node of G on its schedule **do**
4: $\quad round_{res} \Leftarrow 1$ *//First round on all resources*
5: $\quad N_{res} \Leftarrow$ not executed nodes of G in res' schedule with $EFT \leq \frac{makespan}{R/round_{res}}$
6: $\quad$ Send tasks of N_{res} to execute in res.
7: **while** not(all nodes of G have finished) **do**
8: $\quad t \Leftarrow$ wait_next_task_to_finish() *//blocking wait*
9: $\quad res \Leftarrow$ resource of t
10: $\quad avrg_time_{res} \Leftarrow avrg_time_{res} + execution_time_t$
11: $\quad executed_tasks_{res} \Leftarrow executed_tasks_{res} + 1$
12: $\quad$ **if** t is the penultimate task of G in res' schedule **then**
13: $\quad\quad avrg_time_{res} \Leftarrow \frac{avrg_time_{res}}{executed_tasks_{res}}$
14: $\quad\quad$ **if** $avrg_time_{res} > expected_{res} + threshold$ **then**
15: $\quad\quad\quad$ Reschedule not started tasks of res with PCH
16: $\quad\quad\quad res \Leftarrow$ new resource to execute tasks of N_{res}
17: $\quad\quad round_{res} \Leftarrow round_{res} + 1$ *//next round*
18: $\quad\quad N_{res} \Leftarrow$ not started nodes of G in res' schedule with $EFT \leq \frac{makespan}{R/round_{res}}$
19: $\quad\quad$ Send tasks of N_{res} to execute in res.

The case where there is only one task of a DAG on a resource in a round deserves special attention. In this case the scheduler knows that there is performance loss if this unique task does not finish in the expected time. If it finishes in the expected time and there are more tasks from the same graph on the resource schedule, there will be an idle time on the processor until it receives the tasks to the next round. This problem could be solved if the algorithm sends two tasks to a resource, even if the second one should not be sent on that round. Then, if there is loss of performance, the second task (and others on the resource's schedule, if any) could be rescheduled before the end of the first task. If there is no loss of performance, there is no idle period, since the second task could start just after the first one finishes.

Links can suffer of performance loss too. In this algorithm there is no dynamic mechanism to address this issue. The static PCH algorithm uses the bandwidth given by the GM repository, assuming that it is a good forecast, what can be achieved with the Network Weather Service (NWS) [11].

5. EXPERIMENTAL RESULTS

In the chosen strategy, where reschedule is made after an initial static schedule, the results of the dynamic approach are extremely dependent on the initial static schedule. So, it is important to have consistent results for the PCH to give to the dynamic approach a good initial makespan. Thus, first we compared the static PCH algorithm with HEFT and CPOP algorithms, then, we compared the static and dynamic PCH. Fifteen graphs supported by the programming model were randomly taken for the experiments, with number of nodes between 7 and 82. The algorithms were executed with controllers (communication between tasks were via controllers) and without controllers (direct communication between tasks) and with random values in communica-

tion and computation costs. Medium communication means that the communication and computation costs were randomly generated in the same interval. High communication means all communication costs were higher than all computation costs. The experiments were made simulating a group topology, as described in Section 2. For each number of groups, varying from 2 to 25, each graph was scheduled 1000 times. Each group had a random number of heterogeneous resources, varying from 1 to 5.

The main comparison metrics used in the literature are the Schedule Length Ratio (SLR) and the Speedup.
The SLR is given by:

$$SLR = \frac{makespan}{\sum_{n_i \in CP} \frac{instructions_{n_i}}{power_{best}}},$$

where the sum in the denominator represents the computation cost of the task graph critical path on the best resource available. The Speedup is given by:

$$Speedup = \frac{\sum_{n_i \in V} \frac{instructions_{n_i}}{power_{best}}}{makespan},$$

where the sum in the numerator represents the sequential execution time of all tasks in the best resource available. The number of times an algorithm gives a better schedule than another is also a comparison metric.

Figures 3 and 4 show the average SLR and speedup for medium communication. With controllers PCH gives better results than HEFT and CPOP with any number of groups. Without controllers PCH gives results similar to HEFT, with PCH getting better as higher is the number of groups.

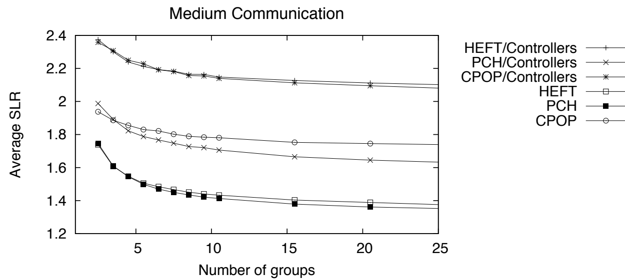

Figure 3: SLR with medium communication.

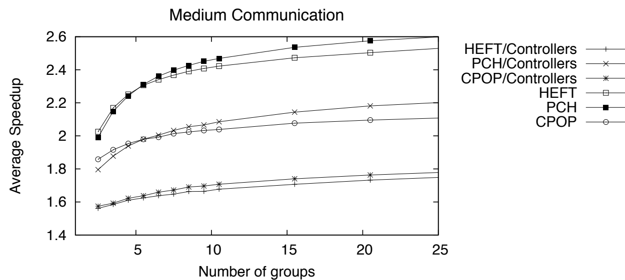

Figure 4: Speedup with medium communication.

In a high communication scenario, the graphics' shapes are similar to that with medium communication, but with increasing advantage for the PCH, since the clusters created by PCH groups coupled tasks, suppressing the communication between them. In a low communication scenario, with controllers PCH is better than HEFT and CPOP, while without controllers it is only better than CPOP. An important aspect of the SLR and speedup results is that, with more than 5 groups, results of PCH *with* controllers are better than results of CPOP *without* controllers. Thus, the clustering strategy adopted by PCH can make the overhead of controllers be low, giving to the infrastructure as a whole recovery, dependability and fault tolerance, provided by the controllers, with high performance.

These experiments indicate that PCH gives good results in a system without variations on resources performance with a group topology. In a grid the resources may not be dedicated, what implies in performance variations. We compared PCH with and without dynamic reschedule. No reschedule means that all tasks are executed in the resources given by the initial schedule, no matter their performance, and reschedule means that the tasks are rescheduled according to the resources performance. 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. The results for executions with a total of 2, 3, and 4 rounds with medium communication and resource performance variation are shown in Figures 5, 6 and 7.

The difference between the results with and without reschedule increases with the number of rounds. For example, with 10 groups and medium communication the absolute differences between the average SLR with and without reschedule are 0.33, 0.69 and 0.93 for 2, 3 and 4 rounds, respectively. For the speedup, these differences are 0.06, 0.13 and 0.18, and for the number of best schedules 2,381; 4,158 and 4,993. In a high communication scenario these differences are wider, as shown in Figures 8, 9 and 10.

The results show that as higher is the communication between tasks, better is the PCH performance. It is explained by the clustering of dependent tasks that suppresses the communication costs. In a performance varying environmnent, the dynamic approach improves the results over the static PCH. The results are better as higher is the number of rounds, since in each round the performance achieved in the execution of tasks is measured, giving a new view of the resources power and providing a reschedule opportunity.

The dynamic PCH can handle the performance loss in resources and the round-based strategy can be, in general, applied to other heuristic-based task scheduling algorithms. Within the Xavantes middleware, the clustering strategy of PCH results in low controllers' communication overhead, providing a recovery mechanism, through controllers, allied to a good performance, as shown by the results.

6. CONCLUSION

This paper presents a dynamic approach for task scheduling in heterogeneous dynamic systems. The algorithm was developed to work with the PCH algorithm and the Xavantes grid middleware, so with support to controllers. For the task graphs supported by Xavantes, the strategy of constructing clusters based on sequential tasks gives good performance when controllers are considered and also when communication is medium or high and controllers are not considered, with time complexity $O(rn^3)$. The dynamic approach deals with performance loss in the resources, sending

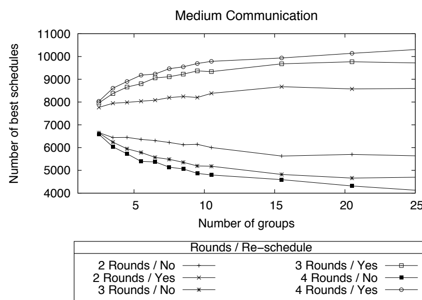

Figure 5: Number of best schedules with reschedule and medium communication.

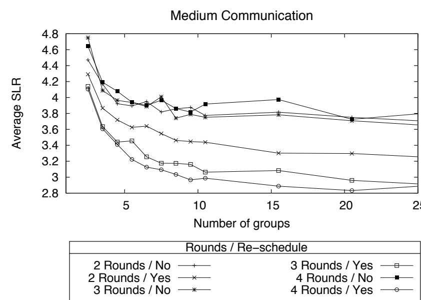

Figure 6: Average SLR with reschedule and medium communication.

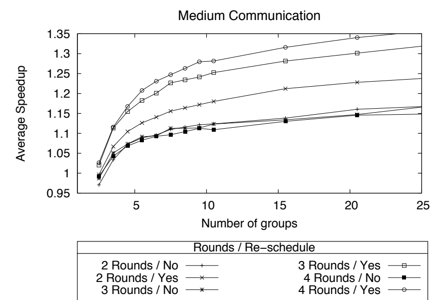

Figure 7: Average peedup with reschedule and medium communication.

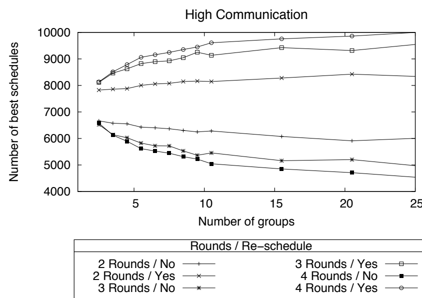

Figure 8: Number of best schedules with reschedule and high communication.

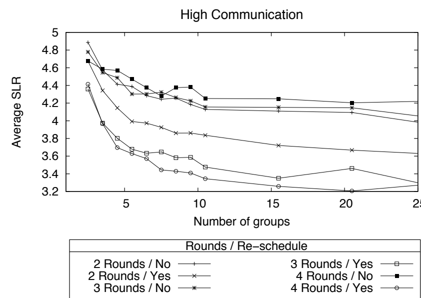

Figure 9: Average SLR with reschedule and high communication.

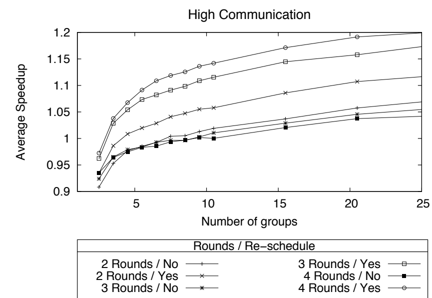

Figure 10: Speedup with reschedule and high communication.

parts of the process to execution at each *round*. As the number of rounds increases, better are the results, since there are more resources measurements during the execution of a process. The introduced concept of *rounds* can be applied to other DAG scheduling systems, adapting them to resources with variable performance.

As future work a study about an ideal number of rounds varying with the depth of the graph could be done. If it is too high, the computation in the rounds can overload the resources. A history of resources performance could be added to the schedule step, trying to do a performance prediction. Also, the issue of links performance loss must be deeply addressed and a reallocation policy could be developed.

Acknowledgements

The authors would like to thank CAPES, FAPESP and CNPq for the financial support.

7. REFERENCES

[1] R. Bajaj and D. P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Trans. Parallel and Distributed Systems*, 15(2):107–118, 2004.

[2] L. F. Bittencourt, E. R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. A path clustering heuristic for scheduling taks graphs onto a grid (short paper). In *3rd ACM International Workshop on Middleware for Grid Computing*, Grenoble, France. nov 2005.

[3] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience*, 18(6):581–594, 2006.

[4] K. Cooper, A. Dasgupta, K. Kennedy, et al. New grid scheduling and rescheduling methods in the grads project. In *18th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.

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

[6] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *2nd Intl. Symp. on Parallel and Distributed Computing, Slovenia*, pages 80–87. IEEE, oct 2003.

[7] T. Hagras and J. Janeček. An approach to compile-time task scheduling in heterogeneous computing systems. In *33rd International Conference on Parallel Processing Workshops*, pages 182–189. IEEE Computer Society, 2004.

[8] 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.

[9] 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.

[10] 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.

[11] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.