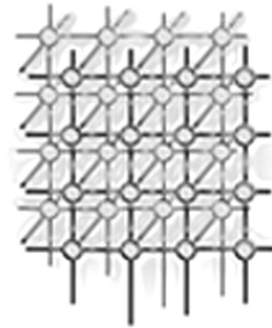
This is a pre-print version.
The final version is available at the publisher's website.



A performance oriented adaptive scheduler for dependent tasks on grids[‡]

Luiz F. Bittencourt[*,1,†] and Edmundo R. M. Madeira[1,†]

[1] *Institute of Computing, State University of Campinas - Brazil*

SUMMARY

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 adaptive 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 adaptive extension changes the size of rounds during the process' execution, taking task attributes and resources performance as parameters, and it can be adopted in other task schedulers. The experiments show that the dynamic *round-based* and the adaptive schedule can minimize the effects of performance losses while executing processes on the grid.

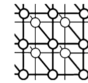
KEY WORDS: *workflow; task scheduling; grid computing*

INTRODUCTION

Grid computing has emerged as a distributed environment with high processing power. Any kind of computational resource can take part in a grid, thus, it is a heterogeneous computing platform that has different link and processor capacities. Furthermore, a grid is in general not dedicated, which 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.

[*]Correspondence to: Institute of Computing, UNICAMP - P.O. 6176, Campinas - São Paulo - Brazil
[†]E-mail: bit@ic.unicamp.br, edmundo@ic.unicamp.br
[‡]This is an extended version of the paper *A dynamic approach for scheduling dependent tasks on the Xavantes grid middleware*, presented at the 4th International Workshop on Middleware for Grid Computing (MGC'06).

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



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 between 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 [4] scheduler. The approach is experimentally evaluated and it shows improvement over a static scheduler. We also propose an adaptive extension with the purpose of making adjustments to the size of the execution queue on each resource as the tasks execute.

This paper is organized as follows. First, a brief introduction to the Xavantes middleware is given, followed by a section with some related works. After that, a section explaining the dynamic approach in conjunction with the PCH algorithm is presented. Next, an adaptive extension is mathematically modeled and described as an algorithm. Finally, experimental results are presented and the paper is closed with the conclusions.

XAVANTES MIDDLEWARE

The 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 [4].

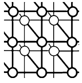
Xavantes' programming model allows to specify applications as structured processes, containing a hierarchy of control constructs to determine the control flow. In a process, an activity represents an atomic task, and a *controller* represents a control structure that organizes the execution order of inner process elements. The controllers can be nested, allowing the hierarchical specification of the control, similarly to structured programming languages. The programming model is based in BPEL, and the controllers are specified in the programming model by the developer. The controllers specification shows to the middleware which tasks are parallel, which tasks are sequential and how the controllers interact.

A *Directed Acyclic Graph* (DAG) is used to represent a process: nodes are tasks, labeled with computational costs; edges are dependency constraints, labeled with communication costs. In the DAG of the 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.

The programming model has some characteristics that restrict on the set of DAGs that can be developed with it. Actually, some DAGs cannot be "directly" developed. Nevertheless, it is possible to use shared variables in the controllers to make DAGs generated by the model to have the same function as any not directly supported DAGs. Also, considering a DAG that cannot be directly developed using the programming model, it is possible to have a task that executes the DAG's subgraph that cannot be written using the programming model. Thus, any

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



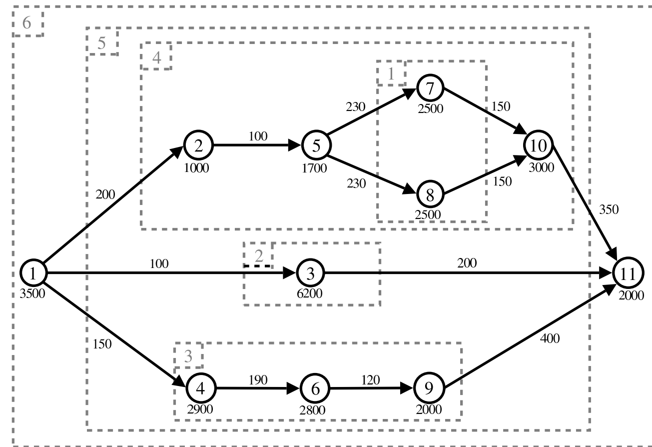


Figure 1: Task graph example.

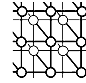
DAG can be executed using the Xavantes middleware, but some of them need to be written in a different way.

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: 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 the 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.

Copyright © 2000 John Wiley & Sons, Ltd.



RELATED WORK

Task scheduling is studied in homogeneous systems ([9, 14]) as well as in heterogeneous systems ([1, 11, 12]). List scheduling, clustering and task duplication are DAG scheduling techniques for homogeneous and heterogeneous systems.

The Heterogeneous Earliest Finish Time (HEFT) [12] 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 $O(rn^3)$ [8], where r is the number of resources and n is the number of tasks of the process. The Critical Path on a Processor (CPOP) [12] 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 [6]. It just sends one task at a time to the scheduler, which schedules the tasks like independent tasks, without knowledge about dependencies. In [10], 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. A dynamic two level scheduling for wide area networks is proposed in [3], 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.

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],[6],[7],[10]) 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. With both dynamic and adaptive proposals, the PCH can adjust the schedule according to the performance of the resources.

A DYNAMIC APPROACH

The objectives of a task scheduler include minimizing the execution time (*makespan*) of a process, maximizing platform utilization and maximizing throughput. In this work we focus on the former. In Xavantes, controllers have an important role in the process execution, since they are responsible for managing the execution and communication of tasks, 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

In [2] we show that the static PCH gives good results in a static heterogeneous environment with a mesh heterogeneous (no-group) topology. 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 on a priority policy. We have chosen the first strategy because it follows a process FCFS (First Come, First Served) policy. This means that the first process to arrive is the first process to end, like in a batch execution of processes. If we choose the second strategy and there are many process arriving, the first process can delay so much that it would be better to execute it locally, not in the grid. The PCH makes 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.
7: end while
```

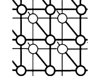
To make a dynamic schedule of tasks we developed the concept of *rounds*. On each round some tasks are selected based on a criterion and sent to execution. Then, the scheduler verifies the performance obtained on each resource used on that round. If the performance of a resource is below a threshold,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

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. With a controller executing on the same resource of most of its tasks, the communication overhead is reduced. For example, in Figure 1 the tasks 4, 6, and 9 are under the controller 3. This way, if tasks 4, 6, and 9 execute in the same resource of controller 3, the communication cost between these tasks would be 0. Note that

Copyright © 2000 John Wiley & Sons, Ltd.



any communication between entities executing in the same resource is 0 because the data does not cross any link.

Definitions

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

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

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

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:***

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

$c_{i,j}$ 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:***

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

P_i is the priority level of the node i.

- ***Earliest Start Time:***

$$EST(n_i, r_k) = \begin{cases} Time(r_k), & \text{if } i = 1 \\ \max\{Time(r_k), \max\limits_{n_h \in pred(n_i)} (EST_h + w_h + c_{h,i})\}, & \text{otherwise} \end{cases}$$

$EST(n_i, r_k)$ 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 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, and selects a processor to it. The necessary information (e.g. available processing power and bandwidth) is given by the infrastructure. Finally, the algorithm schedules the controllers, trying to minimize their communication costs. An overview of these steps is in Algorithm 4 and each step is detailed in the next sections.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

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: $n_{succ} \Leftarrow successor_i$ of n with highest $P_i + EST_i$
5: $cluster \Leftarrow cluster \cup n_{succ}$
6: $n \Leftarrow n_{succ}$
7: **end while**
8: **return** $cluster$

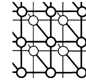
For our example, we use 4 resources in 2 groups. In one group, resources 0 and 1 with processing power of 133 and 130, respectively. In the other, resources 2 and 3, respectively with processing power of 118 and 90. The communication between two resources in the same group is 10 and between two resources in distinct groups is 5. For the graph in Figure 1 and these resources, the first node added to the first cluster, cls_0, is n_1 ($P_1 = 206$), then n_2 ($P_2 = 159.7$), n_5 ($P_5 = 142.2$), n_7, n_{10} ($P_{10} = 72.6$) and n_{11} ($P_{11} = 15$). 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 ($P_4 = 143.9$), which is added to cls_1. Then, n_6 ($P_6 = 103.1$) and n_9 ($P_9 = 70.1$) 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 $cls_3 = \{n_3\}$.

Processor Selection

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.

We could select a processor to a cluster by minimizing the EFT of the last task of the cluster. Another way is to consider also the successor of the cluster's last task, since if we consider only the EFT of the last task of the cluster, the communication cost with its successor could delay the execution. So, 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

Copyright © 2000 John Wiley & Sons, Ltd.



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. 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 scheduling of each cluster, 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:     schedule ⇐ Insert cluster in schedule_r
3:     calculate_EFT(schedule);
4:     time_r ⇐ calculate_EST(successor(cluster))
5: end for
6: return resource r with the smallest time_r
```

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 communication between tasks must be via their controllers, the policy of creating clusters based on sequential tasks in the DAG reduces the communication overhead generated by the controllers. 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. This step is done by the *schedule_controllers()* routine in Algorithm 4, which shows the whole scheduling algorithm. The final schedule for the example is shown in Figure 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 T rounds) if n has not started its execution and if $EFT_n \leq \frac{makespan}{T/k}$ †, with $1 \leq k \leq T$,

† The makespan is the size of the schedule in a given instant. Thus, before the end of the execution, makespan is equal to the estimated finish time of the last task of the process. After the process is finished, the makespan is the real execution time of the process.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

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: $cluster \Leftarrow$ get_next_cluster()
5: $resource \Leftarrow$ get_best_resource($cluster$)
6: Schedule $cluster$ on $resource$
7: Recalculate Weights, ESTs and EFTs
8: **end while**
9: schedule_controllers()

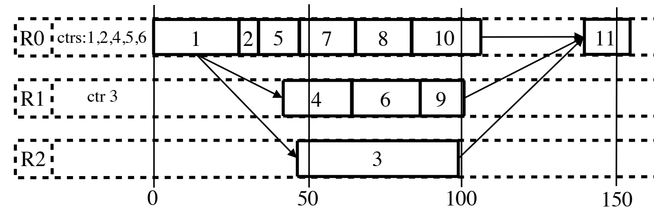
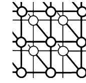


Figure 2: Schedule obtained for the task graph of Figure 1.

or if the task is the first task on the resource schedule. 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 real execution time of the finished tasks on that resource. Then the algorithm compares this real execution time 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 tasks that have not started their execution and are 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.

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)

Copyright © 2000 John Wiley & Sons, Ltd.



Algorithm 5 Dynamic PCH Algorithm

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

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.

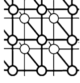
Links can suffer of performance loss too. In this algorithm there is no dynamic mechanism to address this issue. The static PCH algorithm uses the bandwidth given by the GM repository, assuming that it is a good forecast. One way of doing this forecast is with the Network Weather Service (NWS) [13].

ADAPTIVE EXTENSION

After providing a dynamic algorithm to deal with performance losses in resources, now we propose 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:

1. Adapt the size of the rounds according to the weight of the tasks.
2. Adapt the size of each round according to the performance of each resource.
3. Consider the performance history of each resource when deciding the size of the round.

Copyright © 2000 John Wiley & Sons, Ltd.



With this, we developed a mathematical model that shows how these characteristics can be adopted in the scheduling algorithm. Let:

- G be the DAG of the process;
- R be the set of resources selected when scheduling G;
- $M_{r,k}$ be the set of tasks sent to execution on resource r in round k;
- S_r be the schedule of resource r;
- $exp_time_{M_{r,k}} = \sum_{t \in M_{r,k}} (EFT_{t,r} - EST_{t,r})$ be the expected execution time of tasks in $M_{r,k}$.
- Performance of resource r in round $k = p_{r,k} = \frac{exp_time_{M_{r,k}}}{real_exec_time_{M_{r,k}}}$.

We now define the execution time of the resource r in the round $k+1$, $ET_{r,k+1}$, as below:

$$ET_{r,k+1} = \begin{cases} \frac{\sum_{t \in G} weight(t)}{|G|}, & \text{if } k = 0 \\ ET_{r,k} \times p_{r,k} + \lfloor p_{r,k} \rfloor \times rwrd_factor \times exp_time_{M_{r,k}}, & \text{if } k \geq 1 \end{cases} \quad (1)$$

In other words, $ET_{r,k}$ represents the size of the round k in the resource r. The algorithm selects the tasks to send to execution on each round based on this definition, obeying $\sum_{t \in M_{r,k}} weight(t) \leq ET_{r,k}$. Thus, $M_{r,k}$ will be the tasks of G scheduled in r such that the sum of their weights is less than or equal to $ET_{r,k}$. In the case where the weight of the first task to be sent to execution is bigger than $ET_{r,k}$, the first task will be the only one sent to execution on the resource r in the round k ($|M_{r,k}| = 1$).

The multiplication $\lfloor p_{r,k} \rfloor \times rwrd_factor \times exp_time_{M_{r,k}}$ works like a “reward”, increasing the size of the next round each time the resource meets the expected performance. This reward is given in function of the expected execution time for the last round executed on the resource. When $0 < p_{r,k} < 1$, the multiplication is 0, so there is no reward and the multiplication $ET_{r,k} \times p_{r,k}$ decreases the round size. On the other hand, when $p_{r,k} > 1$, the multiplication $ET_{r,k} \times p_{r,k}$ increases the round size and there is also a reward. In the particular case where $p_{r,k} = 1$, the round size is increased only by the reward factor.

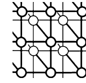
Doing $avrg_w(G) = \frac{\sum_{t \in G} weight(t)}{|G|}$, and expanding the recurrence (1), we have:

$$ET_{r,k+1} \quad = \quad avrg_w(G) \times \prod_{i=1}^{k} p_{r,i} + \sum_{i=1}^{k} \left(\lfloor p_{r,i} \rfloor \times rwrd_factor \times exp_time_{M_{r,i}} \times \prod_{j=i+1}^{k} p_{r,j} \right)$$

Analyzing (2), we can identify the desired properties enumerated above. The first property (adapt the size of the rounds according to the weight of the tasks) is provided by $avrg_w(G)$. The product of performance values is like a history of performance of a resource, giving a view of the achieved performance in the past executions and, in conjunction with the reward factor, it automatically adapts the round size to each resource executing tasks of G. Note that an increase in the round size tends to decrease the number of rounds, and a decrease in the round size tends to increase the number of rounds. An algorithmic view of this strategy is shown in Algorithms 6 and 7.

In line 1 of the Algorithm 6, the DAG is scheduled using the static PCH algorithm. After that, in lines 2 to 7 the algorithm generates the $M_{r,1}$ for all resources that have nodes of

Copyright © 2000 John Wiley & Sons, Ltd.



Algorithm 6 Adaptive Dynamic PCH Algorithm

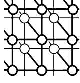
```
1:  Schedule DAG G using the PCH Algorithm
2:  for all r ∈ R do
3:     round_r ⇐ 1 //First round on all resources
4:     next_total_exec_time_r ⇐ avrg_w(G)
5:     M_{r,round_r} ⇐ next_tasks_to_execute(M, r, round_r, schedule_r, exec_time_r, exp_time_r,
       next_total_exec_time_r)
6:     Send tasks of M_{r,round_r} to execution in r.
7:  end for
8:  while not(all nodes of G have finished) do
9:     t ⇐ wait_next_task_to_finish() //blocking wait
10:    r ⇐ resource of t
11:    exec_time_r ⇐ exec_time_r + real_execution_time_t
12:    if t is the last task of G in r's execution queue then
13:       round_r ⇐ round_r + 1 //next round
14:       new_r ⇐ r
15:       if exec_time_r > exp_time_r + threshold then
16:          Reschedule not started tasks of r with PCH
17:          new_r ⇐ new resource selected to execute tasks of schedule_r
18:       end if
19:       M_{new_r,round_new_r} ⇐ next_tasks_to_execute(M, new_r, round_r, schedule_new_r,
          exec_time_r, exp_time_r, next_total_exec_time_r)
20:       Send tasks of M_{new_r,round_new_r} to execute in new_r.
21:    end if
22: end while
```

G on its schedule. The first round on all resources will have the size of the average weight of nodes in G (line 4). Line 5 generates the set $M_{r,1}$ for each resource and then this set is sent to execution in line 6. The loop in line 8 iterates until the process execution finishes. Line 11 accumulates the real execution time of tasks that are finished. Line 12 verifies if the finished task is the last task of the round on the resource. If yes, the algorithm goes to the next round on that resource. Line 15 verifies the necessity of reschedule by comparing the real execution times accumulated in line 11 with the expected execution time, calculated from the DAG attributes. The algorithm generates the new $M_{new_r,round_{new_r}}$ (line 19) and sends it to execution on resource new_r (line 20). Note that new_r can be the same resource as r in two cases: first if r performed as expected, second if the reschedule results in the same schedule (i.e. there is no better resource to execute the remaining tasks).

The Algorithm 7 is the adaptive extension that generates the set $M_{r,k}$ according to the performance of the resources on each round. In lines 2 to 4 the algorithm calculates the performance of the resource on the last round (line 2) and assigns the size of the next round to the variable $next_total_exec_time_r$ (line 3). Line 4 excludes the already executed tasks from the queue. If it is the first round, lines 2 to 4 should not be executed (line 1), since there is no performance to evaluate and there is no already executed tasks on the resource. Then,

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



Algorithm 7 next_tasks_to_execute(M, r, $round_r$, $schedule_r$, $exec_time_r$, exp_time_r, $next_total_exec_time_r$)

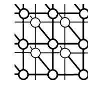
1: **if** $round_r > 1$ **then**
2: $p_r \Leftarrow \dfrac{exp_time_r}{real_exec_time_r}$
3: $next_total_exec_time_r \Leftarrow next_total_exec_time_r \times p_r + \lfloor p_r \rfloor \times rwrd_factor \times exp_time_r$
4: $tasks_to_execute \Leftarrow schedule_r - started_tasks_r$ //set operation
5: **end if**
6: $M_{r,round_r} \Leftarrow$ first_task_of($task_to_execute_r$) //$M_{r,round_r}$ has only the first task of the queue
7: $tasks_to_execute \Leftarrow tasks_to_execute - M_{r,round_r}$ //set operation
8: $exec_time_{M_{r,round_r}} \Leftarrow EFT_t - EST_t$
9: **while** $exec_time_{M_{r,round_r}} < next_total_exec_time_r$ **do**
10: $t \Leftarrow$ first_task_of($tasks_to_execute_r$)
11: **if** $exec_time_{M_{r,round_r}} + (EFT_t - EST_t) \leq next_total_exec_time_r$ **then**
12: $M_{r,round_r} \Leftarrow M_r \cup t$
13: $tasks_to_execute \Leftarrow tasks_to_execute - t$
14: $exec_time_{M_{r,round_r}} \Leftarrow exec_time_{M_{r,round_r}} + (EFT_t - EST_t)$
15: **end if**
16: **end while**
17: return $M_{r,round_r}$

the algorithm assigns the first not executed task to the variable $M_{r,round_r}$. While the sum of the weights of the tasks in $M_{r,round_r}$ is less than the size of the current round (line 9), the algorithm keeps taking the first task of the queue (line 10) and, if there is room to the task (line 11), it is added to $M_{r,round_r}$ (line 12), always accumulating the total weight in the variable $exec_time_{M_{r,round_r}}$ (line 14) and taking the tasks out of the queue (line 13).

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. After that, we compared the dynamic PCH with and without the adaptive extension. 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 communication and computation costs. 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

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls



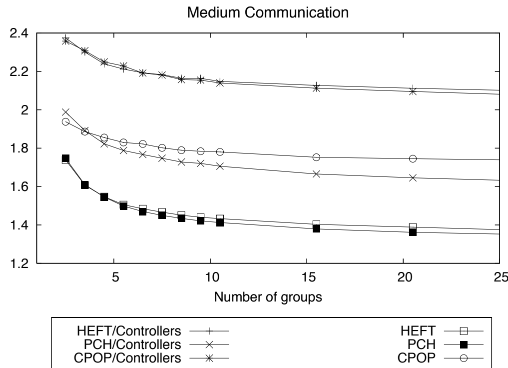


Figure 3: Average SLR.

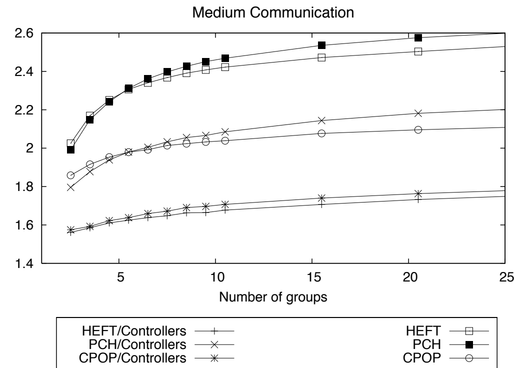


Figure 4: Average Speedup.

experiments were made simulating a group topology, as described in the *Xavantes Middleware* section. 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 shows how many times the schedule length is bigger than the execution of the critical path of the DAG on the best resource available. The speedup shows how many times faster is the execution with the current schedule than the sequential execution of all tasks on the best resource. The number of times an algorithm gives a better schedule than another is also a comparison metric.

Static PCH evaluation

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 when increasing the number of groups.

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. In general, the results with controllers are worse than that without them. This is because controllers generate overhead to provide recovery. However, 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

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.

Dynamic PCH evaluation

The previous 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. To determine the resources performance, we first generated a pattern of resources performance based on some measurements made on computers in our laboratories. Using this pattern, in the beginning of the simulation, we generate processing powers for each task to be executed. These processing powers are the same for the execution of each algorithm. 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 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, for the number of best schedules the differences are 2317, 4624 and 5493. The SLR and Speedup for high communication are shown in Figures 6 and 8. Note that the little difference in the curves without reschedule for 2, 3 and 4 rounds are due to the random parameters used in the simulations. However, what is important to note are the differences between the results for the same number of rounds. These differences are not affected by differences in resources performance, since the same performance values are used on each simulation with the same number of rounds.

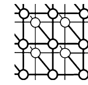
In an environment with performance variation, the dynamic approach improves the results over the static PCH. The results are better as higher is the number of rounds, since in each round the performance achieved in the execution of tasks is measured, giving a new view of the resources power and providing a reschedule opportunity.

Adaptive extension evaluation

After providing evaluations of the static and dynamic PCH, we now evaluate how the proposed adaptive extension behaves. Figures 9 and 10 show the results for the comparison between the dynamic PCH with and without reschedule, and the dynamic PCH with the adaptive extension, in simulations with medium communication. As in the previous experiment, we executed simulations with the dynamic PCH considering performance variations in resources, with and without reschedule, but in these graphics the results for executions with the adaptive

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



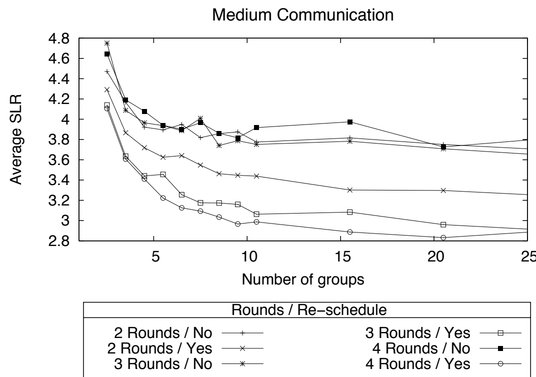


Figure 5: Average SLR with reschedule and medium communication.

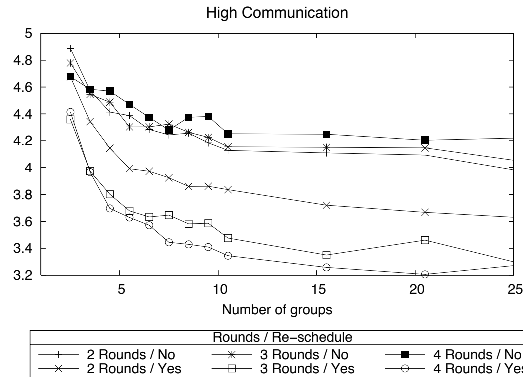


Figure 6: Average SLR with reschedule and high communication.

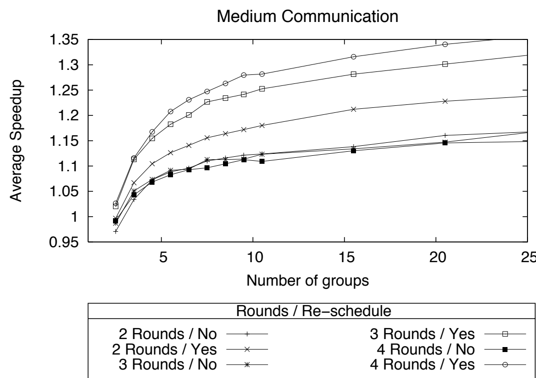


Figure 7: Average speedup with reschedule and medium communication.

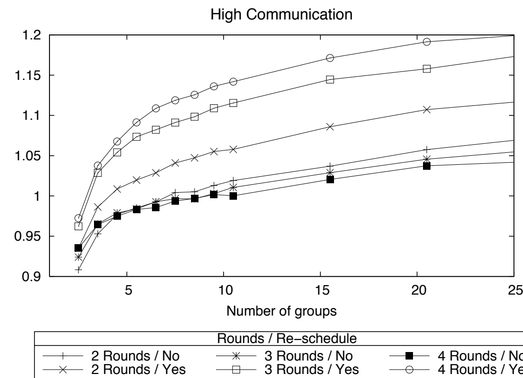


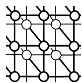
Figure 8: Average speedup with reschedule and high communication.

extension are shown. The number of rounds varies from 2 to 4 in executions of the dynamic PCH without the adaptive extension, and these executions are compared against executions with the adaptive extension. The adaptive algorithm does not have a defined number of rounds. In the figures, the keys T *rounds/Adaptive* have the number of rounds to show that the curve with T rounds of the adaptive extension should be compared with executions of the dynamic PCH with T rounds. The adaptive algorithm was executed with $rwrd_factor = 0.05$. Other simulations performed with different $rwrd_factor$ have shown that if this factor is too high (> 0.30), it can make the performace worse, since it will make the rounds bigger, reducing the number of rounds in a non desirable way. Considering this, the reward factor could be adjusted according to the expectations about the performance of each resource.

The number of best schedules with the adaptive algorithm ranges from $4,000$, for simulations with 2 groups, to $8,000$, for simulations with 25 groups. With the dynamic PCH with reschedule, this range is from $2,000$ to $1,000$. The range without reschedule is from $2,500$

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



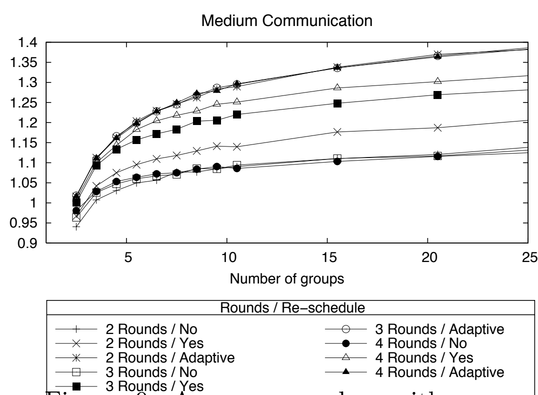


Figure 9: Average speedup with the adaptive algorithm.

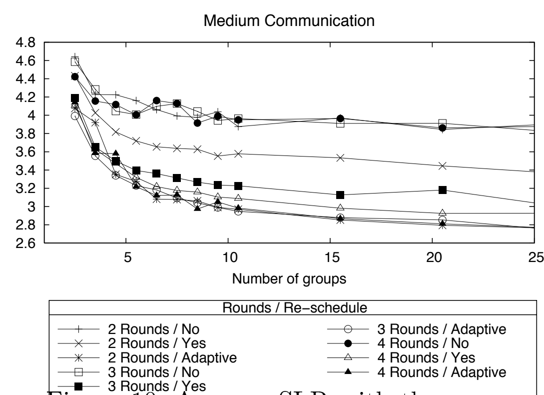


Figure 10: Average SLR with the adaptive algorithm.

to 3,400. The number of best schedules shows that the PCH with the adaptive extension gives more best schedules than the dynamic PCH with reschedule and without reschedule, which suggests that the adaptive algorithm improves the results of the dynamic PCH. We can also conclude that the algorithm with the adaptive extension improves more the results of the dynamic PCH with reschedule than the results without reschedule, because the range of best schedules with reschedule is below the range without reschedule, inverting what is shown in the previous experiment. In spite of this, the dynamic PCH continues to give better average SLR and speedup than the static one, as shown in Figures 9 and 10. A reason for this is because ties are not shown in the number of best schedules, and there are more ties where both the dynamic PCH with reschedule and the adaptive PCH are better than the dynamic PCH without reschedule. Also, as higher is the number of groups, better is the performance of the adaptive algorithm, since there are more options of resources when rescheduling tasks.

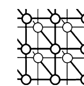
The speedup and SLR results (Figures 9 and 10) show that the adaptive extension can provide faster executions in average when compared to the dynamic PCH without the extension. As expected, as higher is the number of rounds, the dynamic PCH with reschedule tends to approximate its results to that with the adaptive extension, since increasing the number of rounds gives better results to the dynamic PCH with reschedule.

The dynamic adaptive PCH can handle the performance loss in resources and the round-based strategy with the adaptive extension can be 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 experiments.

CONCLUSION

This paper presents a dynamic adaptive approach for task scheduling in heterogeneous dynamic systems. The algorithm was developed to work with PCH and the Xavantes grid middleware,

Copyright © 2000 John Wiley & Sons, Ltd.



providing support to controllers. 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 resources, sending 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 presented adaptive extension regulates the size of the rounds on each resource, using the resources performance as the main parameter, and the introduced concept of adaptive *rounds* can be applied to other DAG scheduling systems, adapting them to resources with variable performance.

As future work the algorithm could be modified to handle parallel scheduling and concurrent process execution, and to deeply address the link performance loss issue. Also, a reallocation policy could be developed to move big tasks that are on resources with poor performance.

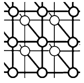
ACKNOWLEDGEMENTS

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

REFERENCES

1. Bajaj R, Agrawal D P. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems* 2004; **15**(2):107–118.
2. Bittencourt L F, Madeira E R M, Cicerre F R L, Buzato L E. A path clustering heuristic for scheduling taks graphs onto a grid (short paper). *Proceedings of the 3rd ACM International Workshop on Middleware for Grid Computing.* Grenoble, France, 2005.
3. Chen H, Maheswaran M. Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems. *Proceedings of the 11th IEEE Heterogeneous Computing Workshop.* Fort Lauderdale, USA, 2002.
4. Cicerre F R L, Madeira E R M, Buzato L E. A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience* 2006; **18**(6):581–594.
5. Cooper K, Dasgupta A, Kennedy K et al. New grid scheduling and rescheduling methods in the GrADS project. *Proceedings of the IPDPS Next Generation Software Program - NSFNGS - PI Workshop.* IEEE Computer Society Press: Los Alamitos, CA, 2004. 199-206.
6. Frey J. Condor DAGMan: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/
7. Fujimoto N, Hagihra K. Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. *2nd International Symposium on Parallel and Distributed Computing.* Ljubljana, Slovenia, 2003; 80–87.
8. Hagras T, Janeček J. An approach to compile-time task scheduling in heterogeneous computing systems. *Proceedings of the 33rd International Conference on Parallel Processing Workshops.* IEEE Computer Society Press: Los Alamitos, CA, 2004; 182–189.
9. Kwok Y -K, Ahmad I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1996; **7**(5):506–521.
10. Prodan R, Fahringer T. Dynamic scheduling of scientific workflow applications on the grid: a case study. *Proceedings of the 2005 ACM symposium on Applied computing (SAC'05).* ACM Press: New York, 2005; 687–694
11. Sakellariou R, Zhao H. A hybrid heuristic for DAG scheduling on heterogeneous systems *Proceedings of the 13th Heterogeneous Computing Workshop.* IEEE Computer Society Press: Los Alamitos, CA, 2004; 111-124.
12. Topcuoglu H, Hariri S, Wu M -Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 2002; **13**(3):260–274.
13. Wolski R, Spring N T, Hayes J. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 1999; **15**(5–6):757–768.

Copyright © 2000 John Wiley & Sons, Ltd.



14. Yang T, Gerasoulis A. DSC: scheduling parallel tasks on an unbounded number of processors *IEEE Transactions on Parallel and Distributed Systems* 1994; **5**(9):951–967.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls